ORM com Aurelius: Conheça esse Framework da TMS

Clube Delphi

Ano 12 :: Edição 142 :: R$ 11,90

DevMedia

BPLs NO DELPHI XE 2

Modularizando sua aplicação

DbExpress de A-Z
Uma visão sobre seu funcionamento

ER/Studio Business Architect XE 2
Modelagem de processos de forma prática

ISSN 1517990-7

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única. Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486

Sumário

Delphi

Boa Ideia
06 – BPLs no Delphi XE 2
[*Paulo Roberto Quicoli*]

Novidade
14 – ER/Studio Business Architect XE 2
[*Fabrício Hissao Kawata*]

Delphi
22 – ORM com Aurelius
[*Giuliano Scombatti Pinto*]

Minicurso
32 – Desenvolvimento Android – Parte 2
[*Giuliano Scombatti Pinto*]

Easy
42 – dbExpress de A a Z
[*Alexandre Borges*]

Olá, eu sou o **DevMan**! Desta página em diante, eu estarei lhe ajudando a compreender com ainda mais facilidade o conteúdo desta edição. Será um prazer contar com sua companhia! Confira abaixo o que teremos nesta revista:

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

RENOVE JÁ!

Sua assinatura pode estar acabando

Renovando a assinatura de sua revista favorita você ganha brindes e descontos exclusivos.

www.devmedia.com.br/renovacao ou (21)3382-5038

DevMedia

Clube Delphi

Ano 12 - 142ª Edição 2012 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral

Paulo Quicoli
pauloquicoli@gmail.com

Editor Técnico

Daniel Sobrinho Laporte
daniel.laporte@gmail.com

Equipe Editorial

Guinther Pauli
guintherpauli@gmail.com

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br

Giuliano Scombatti Pinto
giuliano@sygnux.com.br

Jornalista Responsável

Kaline Dolabella - JP24185

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/mancad
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Paulo Quicoli - Editor da Revista
pauloquicoli@gmail.com

EDITORIAL

A TMS lançou recentemente um framework que muitos desenvolvedores esperavam, o Aurelius. Este é um framework ORM que facilita a utilização da orientação a objetos na prática. Ele traduz de forma fácil os objetos para banco de dados e vice versa. O Giuliano apresenta essa ferramenta desenvolvendo um aplicativo que mostra alguns dos pontos fortes do framework.

Eu particularmente apresento um artigo sobre modularização através do uso de BPLs. Se você nunca utilizou esse recurso ou pensa que uma BPL é apenas para componentes, veja nesse artigo que você pode fazer mais com elas. Esse artigo foi solicitado pelos leitores que participaram de uma enquete lançada por nossa equipe no Facebook. Se você ainda não deu sua opinião, acesse nossa página no Facebook e participe.

Temos um novo colaborador, o Alexandre. Ele é instrutor oficial da Embarcadero e traz para nós um artigo sobre dbExpress, acredito que ao longo desse assunto muitas dúvidas serão respondidas e dicas poderão ser aproveitadas.

O Fabrício continua nos apresentando as ferramentas que formam o Ecossistema Embarcadero, agora falando sobre o ER/Studio Business Architect XE 2 e como ele pode auxiliar na modelagem de negócio de sua empresa. É muito importante possuir essa visão de negócio e não apenas o olhar técnico. Entendendo mais sobre um negócio específico podemos identificar o que gera mais valor agregado ao cliente e dar prioridade a isso durante o desenvolvimento. O Fabrício ainda apresenta o ASP.NET MVC, mostrando pontos fortes de sua utilização, seus benefícios e como você desenvolvedor Prism pode se beneficiar com isso.

Por fim, temos com o Giuliano a segunda parte do minicurso sobre mobilidade, que desta vez apresenta a criação da camada servidora com DataSnap, gerando respostas no formato JSON e um artigo sobre SCRUM do nosso colega Renato, que fundamenta algumas de suas atividades.

Boa leitura!

Paulo Quicoli
pauloquicoli@gmail.com
Editor Geral
quicoli.wordpress.com
twitter.com/pauloquicoli

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia:
www.devmedia.com.br/mvp

DevMedia

Dificuldades em encontrar conteúdo sobre .Net para iniciantes?

A DevMedia simplifica pra você!

easy .net magazine

Easy .Net Magazine:

A primeira revista digital feita especialmente para quem está dando os primeiros passos em .Net!

Você encontrará exemplos e modelos fáceis e práticos, noções introdutórias e conceitos básicos.

Conheça as vantagens:

- Vídeos e apresentações "embutidos" nas páginas;
- Ilustrações, diagramas e modelos apresentados em PowerPoint;
- Entrevistas em vídeo;
- Códigos-fonte que podem ser copiados com CTRL+C/CTRL+V.

A Easy.Net não é feita em PDF ou Flash!
Ela utiliza um aplicativo 100% online.

DevMedia group

Acesse agora a edição Nº 1 totalmente grátis e comprove

www.devmedia.com.br/easynet

Easy.Net Magazine, seu curso inicial de .Net!

BPLs no Delphi

Modularize sua aplicação

Desenvolver uma aplicação em pacotes (packages) do Delphi é um padrão pouco utilizado entre os desenvolvedores, mas veremos ao longo do artigo que esta prática pode proporcionar várias vantagens. Quando escrevemos e compilamos uma aplicação Delphi, tipicamente é gerado um único arquivo executável que não depende de ninguém para funcionar. Para entender como que uma modularização pode impactar ao criar um novo aplicativo VCL no Delphi XE 2, salva-lo da forma que foi criado e executá-lo, teremos um executável de 6.75MB. Isso mesmo, quase 7MB. Porém podemos alterar a forma como um executável é gerado, ativando o parâmetro Link with Runtime Packages do menu Project> Options>Packages>Runtime Packages, como mostra a **Figura 1**.

Nota

A localização do menu apresentado contempla as versões XE e XE 2 do Delphi

Agora o executável gerado possui apenas 2.18MB. O valor padrão do parâmetro de compilação Link with Runtime Packages é false, assim, sempre que construímos uma aplicação Delphi o compilador adiciona no executável gerado todo o código que é necessário para que o mesmo possa ser executado. Por isso o executável é grande, o uso de pacotes (packages) não está sendo empregado.

Resumo DevMan

De que se trata o artigo:

Neste artigo abordamos a criação de BPLs e consequentemente a modularização que pode ser obtida com seu uso. Uma BPL é um recurso do Delphi que permite centralizar recursos (procedimentos, funções, classes, tipos, etc) que podem ser compartilhados por mais de um projeto. Veremos as diferentes formas de se carregar uma BPL e as vantagens de seu uso.

Em que situação o tema é útil:

A criação de BPLs é útil pois, pois podemos reutilizar código entre projetos, além de que em sistemas grandes, pode se modularizar a carga de recursos. Por exemplo, se criarmos BPLs de acordo com as funcionalidades de um sistema, podemos carregar somente as que estarão em uso, de acordo com o perfil de um usuário logado, ou setor da empresa onde o mesmo está instalado. Outro ponto é que ao dividir um projeto em BPLs, estamos diminuindo o tamanho do executável final, o que facilita em uma atualização de sistema por exemplo (pois menos bytes serão transferidos até o cliente).

Modularização com BPLs:

Uma BPL é conceitualmente muito similar a uma DLL, mas é destinada a sistemas Delphi. É utilizada para centralizar recursos compartilhados, almejando uma reutilização de código e de recursos. Neste artigo abordamos a criação de BPLs, os tipos linkagem oferecidos pelo Delphi, os tipos de carga de BPLs, como inserir formulários nessas BPLs e consumir esses formulários .

Definindo package

A definição para package é bem simples. Ela nada mais é do que um tipo especial de DLL que é utilizada por aplicações Delphi e pelo próprio IDE. Não é algo novo, o conceito de packages existe desde a versão 3 do Delphi.

Seu uso torna possível alocar partes de nossa aplicação em módulos separados que podem ser compartilhados com outras aplicações. Outro cenário de vantagem é quando apenas uma pequena parte da aplicação sofre uma atualização, uma correção de bug por exemplo. Se o conceito de packages está em uso, é possível atualizar em seu cliente somente a package necessária e não toda a aplicação.

Existem basicamente dois tipos de packages:

- Runtime – funciona como DLLs padrão, suas funcionalidades estão acessíveis durante a execução do aplicativo;
- Design – utilizada para instalação de componentes no IDE do Delphi e para criação de editores de propriedades diferenciados. Este tipo de package é utilizado apenas pelo Delphi e nunca é distribuído em uma aplicação.

DLLs vs. BPLs

Uma DLL centraliza código que pode ser utilizado por vários aplicativos. É exatamente isso que a torna poderosa, quando retiramos códigos de aplicativos e os colocamos em uma área compartilhada que outros podem acessar. Sem dúvida alguma o melhor exemplo de uso de uma DLL está na própria API dos Windows que é composta por várias DLLs.

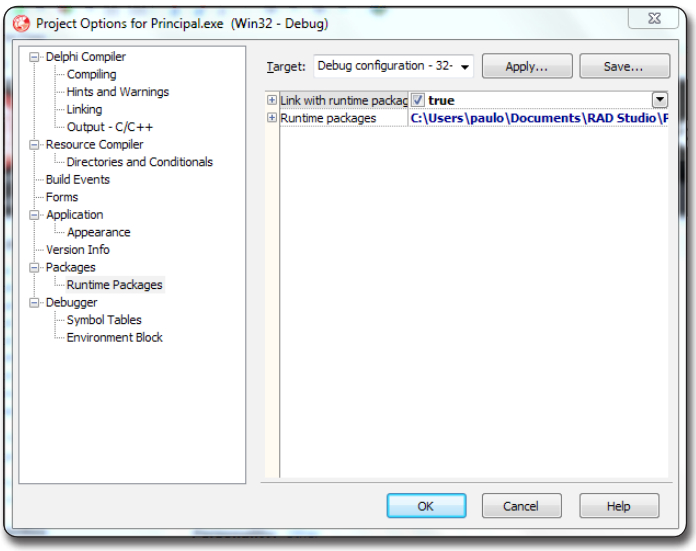


Figura 1. Opção Link with Runtime Packages

É possível inserir em uma DLL um formulário Delphi e reutilizá-lo, outro uso comum é armazenar arquivos de recursos como imagens. Mas antes de comparar uma DLL com uma BPL temos que entender as duas maneiras de se linkar código em um executável: de forma estática ou dinâmica.

Nota do DevMan

Quando programadores conversam sobre a criação de um programa, é comum ouvir a expressão "o sistema está compilado" ou, quando são perguntados se um determinado programa está pronto, respondem "vamos compilar e ver". Esses são termos muito comuns que podem causar confusão na cabeça de programadores iniciantes. Compilação não é o mesmo que criar um código executável. Criar um executável é um processo que utiliza dois componentes: compilação e linkagem. Mesmo que um programa compile, ele pode não funcionar devido a erros na fase de linkagem. A linkagem refere-se à criação de um arquivo executável a partir de vários arquivos .DCU. Esses são gerados pela fase de compilação e contêm informações em formato binário. A fase de compilação verifica a sintaxe (escrita dos comandos) de cada unit (.PAS) e para aquelas que estão corretas, gera seu respectivo DCU (Delphi Compiled Unit).

Estática: é quando um projeto Delphi é compilado e todo o código requerido por essa aplicação está diretamente linkado ao executável. O executável gerado contém o código de todas as units envolvidas no projeto. Com certeza você pensou: "todas as units que referencio na seção uses então são adicionadas ao executável?". Na verdade não. O compilador Delphi é esperto o suficiente para detectar o que está em uso ou não e somente incorporar o que realmente interessa, adicionando o mínimo de código possível. Este é o comportamento padrão da compilação Delphi.

Dinâmica: funciona como quando se trabalha com DLLs. Isto é, as funcionalidades são adicionas à aplicação, mas sem inserir o código diretamente. Quando parte de um código que consta em um package é chamado, este é então carregado. Essa carga acontece de forma automática, não é necessário codificar nada para isso.

Os packages diferem das DLLs porque eles são específicos para o Delphi, isto é, aplicações escritas em outras linguagens não podem utilizar os packages criados em Delphi.

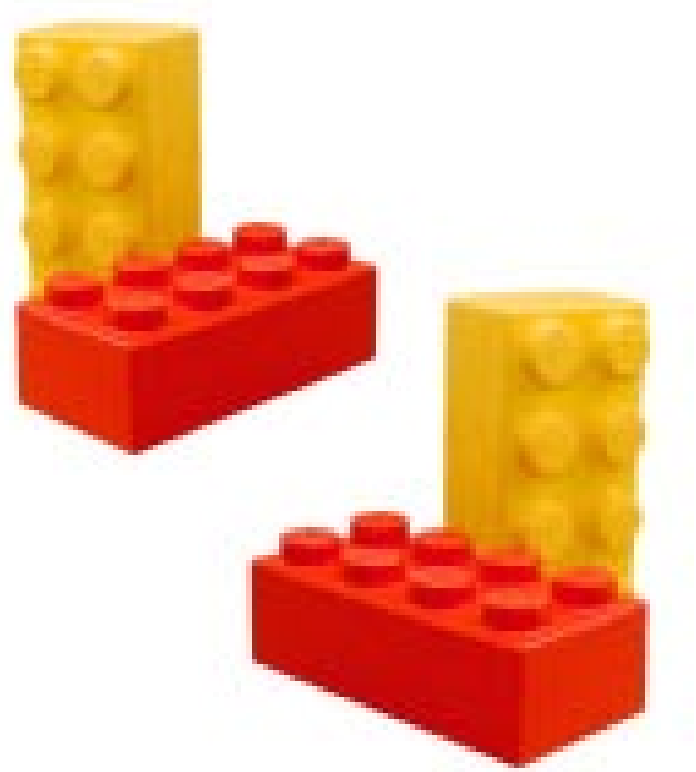
De uma maneira geral quando criamos uma DLL é para que ela armazene rotinas que possam ser utilizadas pelos mais diversos sistemas. Um package vai além disso. Ele pode conter código, componentes, formulários e até mesmo classes Delphi.

A verdade é que ambos, DLLs e packages, auxiliam na redução de código. A primeira razão por usar um mecanismo como este está na redução do tamanho das aplicações. Mas esse ponto deve ser visto de uma forma mais ampla. Não é porque você está quebrando sua aplicação em packages que você não vai enviá-los ao cliente. Outro ponto que pode passar despercebido é que packages economizam memória. Isso é resultado da linkagem dinâmica, onde somente uma cópia da VCL reside em memória para todas aplicações Delphi que utilizam packages de runtime.

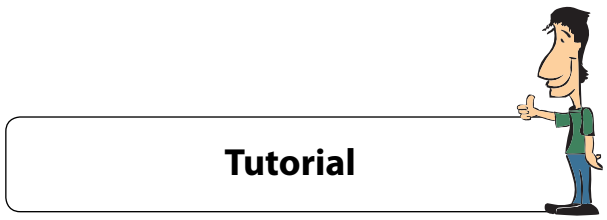
Versionamento

Quando queremos atualizar uma DLL, seja porque adicionamos algo novo ou alteramos a implementação de alguma função, simplesmente compilamos e distribuímos a nova DLL. Todas as aplicações que a utilizam continuarão funcionando, isso se não removemos alguma função que deveria ser utilizada.

Em relação aos packages, quando atualizamos um, não podemos simplesmente distribuí-lo sem atualizar o executável. Como poderá ser visto, um package não é nada mais que uma coleção de units. Todos arquivos DCU contém essa informação.



É justamente por isso que não podemos usar uma unit compilada no Delphi 7 no Delphi XE 2, a não ser que tenhamos o código fontes dessa unit. O compilador verifica o número de versão e decide se precisa recompilar os fontes. Por isso não é possível utilizar um package do Delphi XE 2 no Delphi 7 e vice versa. Sempre que há troca de versão a recompilação é forçada. É justamente por isso que é uma prática comum incluir a versão no nome de packages, por exemplo "HelpDP_70", onde 70 indica que o package é para a versão 7 do Delphi, evitando assim confusões.



Sua primeira BPL

Ao abrir o Delphi XE 2 temos uma opção exclusiva para isso. No menu File>New encontramos Package (**Figura 2**).

Esta opção adiciona no Project Manager um novo projeto chamado Package1.bpl. Nele temos duas pastas: Contains e Requires, como mostra a **Figura 3**.

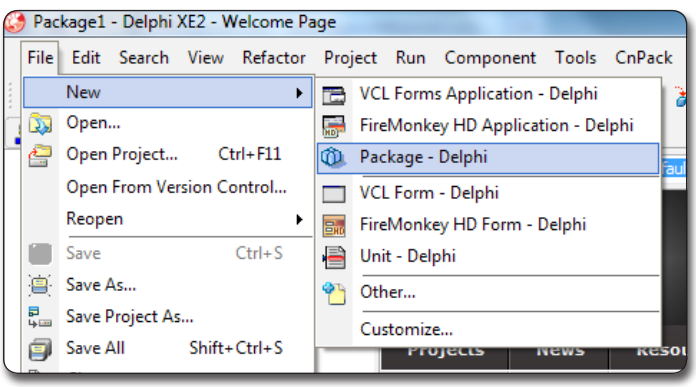


Figura 2. Menu package

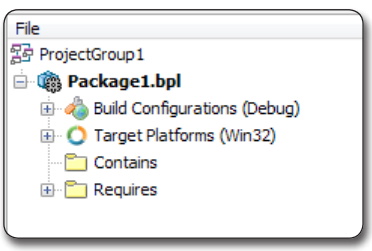


Figura 3. Pasta Contains e Requires

A pasta Contains é utilizada para armazenar os arquivos (units) que criamos, enquanto que Requires representa suas dependências. Para efeito de demonstração vamos adicionar uma nova Unit, clicando com o botão direito do mouse sobre Contains e escolhendo Add New>Unit, e nela adicionar o código da **Listagem 1**. Agora podemos definir se este será um package de runtime ou de design. Acessando o menu Options da Package1.bpl, na opção Description temos as opções Usage Options. Vamos escolher apenas Runtime only. Isso define que nosso package é de runtime, ou seja, não pode ser "instalado" como se faz com um componente comum.

Então podemos salvar tudo agora, para a unit criada deixe o nome sugerido, já para o projeto salve como MinhaBPLXE2. Clicando com o botão direito do mouse sobre o projeto MinhaBPLXE2 agora você tem a opção Compile. Compile e veja que o Delphi identificou que será necessário incluir outro package na pasta Requires. Apenas confirme a solicitação do IDE do Delphi.

Listagem 1. Exibindo mensagem

```
procedure ExibirMensagem(texto: string);

implementation

uses Vcl.Dialogs;

procedure ExibirMensagem(texto: string);
begin
 ShowMessage(texto);
end;
```

Utilizando a primeira BPL

Para testar nosso package vamos criar um novo projeto VCL. No formulário principal simplesmente acrescente um botão. O evento OnClick desse irá chamar a função ExibirMensagem definida no package.

Acessando o menu Options desse projeto podemos localização a opção Packages>Runtime Packages. Ali marcamos a opção Link with runtime packages e editamos o campo Runtime Packages para localizar nossa BPL e adicioná-la ao projeto.

Mesmo que estejamos adicionando a BPL ao projeto, quando formos utilizá-la é preciso listar na seção uses suas units. Isto porque realizar a compilação com runtime packages apenas diz para a aplicação onde encontrar os fontes dos componentes. Sendo assim, o evento OnClick pode ser codificado como mostra a **Listagem 2**.

Listagem 2. Utilizando a BPL

```
uses Unit1;

{$R *.dfm}

procedure TForm2.Button1Click(Sender: TObject);
begin
 ExibirMensagem('Mensagem exibida a partir da BPL');
end;
```

Nota

Ao distribuir uma aplicação marcada para utilizar runtime packages, é muito importante assegurar que o usuário final tenha além do executável, todas as BPLs necessárias, caso contrário o aplicativo não irá funcionar. Dê preferência para deixar essas BPLs no próprio diretório do aplicativo. Este é o primeiro local que será procurado.

O que fizemos neste primeiro momento foi a linkagem estática. Nós indicamos nas configurações do projeto qual BPL carregar. Esta é a forma mais simples de uso, mas não podemos esquecer que temos também a linkagem dinâmica.

Utilização dinâmica

O Delphi oferece dois comandos que permitem a carga dinâmica de BPLs, o LoadPackage e o UnloadPackage que respectivamente carregam e descarregam um package da memória. Uma vez carregado, a aplicação localiza todas as funções exportadas, ficando assim prontas para o uso.

Mesmo se formos optar pela utilização dinâmica ainda é preciso marcar a aplicação para utilizar packages runtime. O detalhe é que os packages relacionados no campo serão linkados de forma estática. Se vamos carregar alguma BPL dinamicamente, não devemos relacioná-la nessa lista, para esse caso é uma boa prática deixar ali somente os packages padrão da VCL, o restante que nós criarmos será carregado por demanda.

Alguns benefícios

Quando um package não está disponível, uma aplicação que usa a linkagem estática simplesmente para de funcionar, enquanto que em uma que utiliza a forma dinâmica pode tratar esse erro e continuar funcionando. Outro cenário que tira vantagem da linkagem dinâmica é o carregamento de formulários. Digamos que uma aplicação possua mais de 50 formulários e dependendo de alguma configuração no sistema, um determinado usuário só tenha acesso a 20. É melhor armazenar e organizar esses formulários em pacotes e carregar somente o necessário. Outra vantagem está no fato de que uma aplicação que carrega packages dinamicamente irá iniciar mais rapidamente porque somente o código necessário para a carga do aplicativo é carregado.

Utilizando um formulário dinâmico

Para vermos como o Delphi utiliza o carregamento dinâmico de packages, vamos criar uma BPL e nela incluir um formulário que será acessado por um projeto principal. Para começar vamos criar um novo projeto VCL. Ao Project Group criado vamos adicionar um segundo projeto, desta vez do tipo Package. Após salvar tudo, clicamos com o botão direito na pasta Contains da BPL e escolhemos Add New>VCL Forms.

O Delphi irá detectar que falta incluir na pasta Requires uma referência à VCL e irá exibir um diálogo pedindo confirmação. Apenas confirmamos e o formulário é apresentado.

Este é um formulário como qualquer outro. Podemos inserir componentes, configurar sua aparência e dar-lhe um nome adequado. A **Figura 4** mostra seu visual.

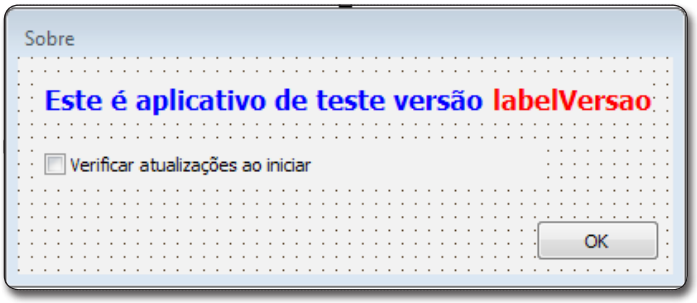


Figura 4. Formulário Sobre

Quando formos utilizar esse formulário vamos passar a ele a versão atual do projeto e obter em resposta um valor booleano, indicando se o usuário marcou o devido CheckBox ou não.

Vamos lembrar que quando utilizamos a carga dinâmica, nós não referenciamos na seção uses a origem (unit) das rotinas ou formulários que vamos utilizar. Assim, o código a seguir não compilaria no projeto principal:

```
SobreF:=TSobreF.Create(Application);
try
 SobreF.DoSomething;
finally
 SobreF.Release
end;
```

O compilador não reconheceria a variável e tipo SobreF, TSobreF. Para resolver esse problema nós providenciamos dentro da BPL um método que se encarrega de instanciar, exibir, liberar da memória o formulário criado, retornar um valor e ainda receber um valor. Esse será o método Executar visto na **Listagem 3** e que está localizada na seção implementation do formulário da package.

Listagem 3. Método Executar

```
function Executar
  (const versao: string): boolean;
begin
  Result:=False;
  SobreF:= TSobreF.Create(nil);
  try
    SobreF.labelVersao.Caption := versao;
    SobreF.ShowModal;
    Result := SobreF.chkVerificarAtualizacoes.Checked;
  finally
    FreeAndNil(SobreF);
  end;
end;
```

Agora, como um package nada mais é do que uma DLL melhorada é preciso exportar essa função, assim outros aplicativos poderão chamá-la dinamicamente. Isso é feito com o uso da palavra reservada exports. Ela deve ser posicionada ao fim da unit do formulário, antes do último end, como é mostrado a seguir:

```
//...
exports
  Executar;

end.
```

É muito importante frisar que os nomes dos métodos dentro do bloco exports seguem o padrão case sensitive, ou seja, Executar é diferente de executar. Assim, os métodos devem ser grafados exatamente como foram declarados. Feito isso podemos gerar a BPL, compilando seu código fonte, antes lembrando claro de marcá-la como do tipo Runtime Only.

Realizar a chamada de rotinas em packages dinamicamente carregados infelizmente não é algo tão simples. É preciso seguir alguns passos para que tudo funcione corretamente. Em primeiro lugar adicionamos ao projeto principal um botão e em seu evento Click vamos exibir o formulário dinâmico. Essa chamada é feita através da função LoadPackage. Ela tem o objetivo de carregar um package que lhe é passado, neste exemplo assumimos que a BPL está na mesma pasta do executável principal. LoadPackage retorna então um valor do tipo HModule. Esse tipo é um Handle para um módulo Windows, um Handle por sua vez é um identificador numérico único que o Windows utiliza para localizar seus recursos. Se este retorno for válido, ou seja, diferente de 0 (zero), podemos prosseguir pois o package foi localizado.

Como temos o handle (identificador) do package podemos utilizar outra função para extrair os métodos exportados, a função GetProcAddress. Ela recupera o endereço de funções exportadas em bibliotecas dinâmicas, sejam elas BPLs ou DLLs. O primeiro parâmetro dessa função é justamente o handle do package que será verificado, o segundo é o nome da função que queremos acessar. Se a função for encontrada podemos chamá-la e finalmente descarregamos a BPL da memória. Tudo isso pode ser visto na **Listagem 4** a seguir.

Nota

Clique com o botão direito sobre o projeto da BPL e acesse o menu Options. Na tela de opções que se abre temos Delphi Compiler. As propriedades disponíveis configuram o comportamento do compilador Delphi. Ajuste sua propriedade Package Output Directory para ".\$(Platform)\$(Config)", assim o arquivo .BPL será gerado na mesma pasta dos arquivos compilados, facilitando seu acesso.

Listagem 4. Chamando o formulário dinâmico

```
01 procedure TPrincipalF.btnSobreClick(Sender: TObject);
02 var
02  minhaPackage: HModule;
04  FuncaoExecutar: function(const versao: string): boolean;
05  versao: string;
06 begin
07   versao := '1.0';
08   minhaPackage:=LoadPackage(ExtractFilePath(ParamStr(0))
09              + 'FormAuxiliarXE2.bpl');
10  if minhaPackage <> 0 then
11  try
12    @FuncaoExecutar:=GetProcAddress(minhaPackage,'Executar');
13    if Assigned(FuncaoExecutar) then
14    begin
15      if FuncaoExecutar(versao) = True then
16        ShowMessage('Vamos atualizar ao iniciar')
17      else
18        ShowMessage('Não vamos atualizar ao iniciar');
19    end
20    else
21    begin
22      ShowMessage ('Função executar não foi encontrada');
23    end
24  finally
25    UnloadPackage(minhaPackage);
26  end
27  else
28    ShowMessage ('BPL FormAuxiliarXE2 não foi encontrada');
29 end;
```

A linha 4 pode parecer um pouco estranha. Ela faz uso de um tipo especial, o tipo procedural. Um tipo procedural permite que procedimentos e funções possam ser representados em variáveis, que por sua vez podem ser passadas a outras funções ou procedimentos. A declaração de um tipo procedural deve ter a assinatura da função/procedimento que se deseja representar, observe que a declaração usada possui a mesma forma da função que foi definida na BPL.

Fizemos o acesso a um formulário através um método exposto na BPL. Também é possível acessar a própria classe do formulário criado. Para isso ser possível é preciso utilizar a função

RegisterClass para registrar uma classe, no caso o formulário, e a UnRegisterClass para retirar o registro. Respectivamente devem ser chamadas na seção inicialization e finalization, como mostra o código a seguir da BPL.

```
initialization
 RegisterClass(TSobreF);

finalization
 UnRegisterClass(TSobreF);
```

Nota do DevMan

A assinatura de um método nada mais é do que o formato de sua declaração, a quantidade de parâmetros definidos e o tipo de dado que é retornado.

Consequentemente a maneira de acessar o formulário pelo projeto principal também é alterada. Para isso adicionamos um segundo botão nele e codificamos seu evento OnClick como é mostrado na **Listagem 5**.

Listagem 5. Acessando diretamente uma classe

```
01 procedure TPrincipalF.btnSobre2Click(Sender:TObject);
02 var
03  minhaPackage:HModule;
04  versao:string;
05  APersistentClass:TPersistentClass;
06  sClassName:string;
07  AForm:TForm;
08 begin
09  versao := '2.0';
10  sClassName := 'TSobreF';
11  minhaPackage:=LoadPackage(ExtractFilePath(ParamStr(0))
12            + 'FormAuxiliarXE2.bpl');
13  if minhaPackage <> 0 then
14  try
15   APersistentClass := GetClass(sClassName);
16   if APersistentClass = nil then
17    raise Exception.Create('Classe não localizada!');
18   if APersistentClass.ClassParent = TForm then
19   begin
20    AForm := TComponentClass(APersistentClass).
21    Create(nil) as TForm;
22    AForm.ShowModal;
23    FreeAndNil(AForm);
24   end;
25  finally
26   UnloadPackage(minhaPackage);
27  end;
28 end;
```

Para que o projeto principal possa acessar alguma classe existente da BPL ele precisa saber o nome dessa classe. Por isso temos a variável sClassName recebendo o valor "TSobreF". A referência da classe é obtida pelo método GetClass que recebe como parâmetro o nome de alguma classe. Com a referência passada, podemos então realizar um cast para sua classe base TForm, instanciar e exibir a mesma.

Você pode ter notado que dessa vez o formulário não abriu exibindo o número da versão e nem retornou se o checkbox estava marcado. Isso acontece porque estamos acessando o formulário através de uma classe base, TForm, consequentemente não temos acesso às propriedades específicas do formulário TSobreF. Como resolver essa questão então?

Organizando dependências

Criar BPLs é só a ponta de um processo chamado modularização. É preciso pensar nas dependências existentes e até mesmo criar algumas dependências para que possamos atingir nossos objetivos. Neste exemplo o objetivo é ter acesso a algumas informações da classe TSobreF, para podermos passar a ela o número da versão a ser exibido e até mesmo obter qual a opção escolhida pelo usuário.

Vamos aplicar o conceito de package comum, uma BPL que contém as definições necessárias para o projeto principal e que podem ser utilizadas como base para as BPLs dinâmicas. No projeto principal esse package será linkado estaticamente, enquanto que para o package ele estará presenta como um requerimento, na pasta Requires.

Criamos então um novo package no Project Group, chamando-o de FormSobreBaseXE2. Nele adicionamos um novo formulário, sem visual diferenciado, apenas contento os seguintes métodos públicos vistos na **Listagem 6.**

Listagem 6. Métodos base

```
 TFormSobreBaseF = class(TForm)
private
public
  procedure SetaNumeroVersao(Numero: string); virtual; abstract;
  function UsuarioMarcouAtualizao: Boolean; virtual; abstract;
end;
```

A classe base expõe os métodos SetaNumeroVersao e Usuario MarcouAtualizacao com os modificadores virtual e abstract. Virtual indica que o método em questão poderá ser sobrecarregado em alguma classe descendente, já abstract determina que o método não tem implementação na classe onde é declarado e força que suas subclasses o implementem. Selecione que esse é um package de runtime e compilamos. Agora voltamos nossa atenção ao package FormAuxiliarXE2. Para dizer que este package necessita de outro, é preciso adicionar uma referência à pasta Requires. Clicando com o botão direito nela temos a opção Add Reference. Nessa opção localizamos a BPL comum, FormSobreBaseXE2, e confirmamos.

Foi adicionado um segundo formulário, apenas para não confundir com o já existente, de nome Sobre2F, nele alteramos sua classe base de TForm para TFormSobreBaseF e na seção uses acrescentamos FormSobreBaseU. Nesse formulário nós criamos um visual semelhante ao sobre já existente, mas diferença mesmo é que temos que implementar os métodos abstratos definidos em TFormSobreBaseF, como mostra a **Listagem 7**.

Listagem 7. Implementação de métodos

```
//na declaração da classe
procedure SetaNumeroVersao(Numero: string); override;
function UsuarioMarcouAtualizao: Boolean; override;

//implementação
procedure TSobre2F.SetaNumeroVersao(Numero: string);
begin
 inherited;
 labelVersao.Caption := Numero;
end;

function TSobre2F.UsuarioMarcouAtualizao: Boolean;
begin
 Result := chkVerificarAtualizacoes.Checked;
end;
```

Ao aplicar o modificador override estamos dizendo ao Delphi que estamos implementando o método. Um detalhe importante, como uma nova classe criada e vamos acessar essa classe de forma dinâmica, também é preciso utilizar os métodos RegisterClasse e UnRegisterClass respectivamente na seção initialization e finalization, só que registrando agora a classe TSobre2F.

Feito isso falta dizer ao projeto principal para utilizar nosso package comum. Isso é feito acessando o menu Options do projeto e na opção Packages>Runtime Packages incluir uma referência a FormSobreBaseXE2.

Para ter uma melhor visualização do que está acontecendo a **Figura 2** exibe o relacionamento criado.

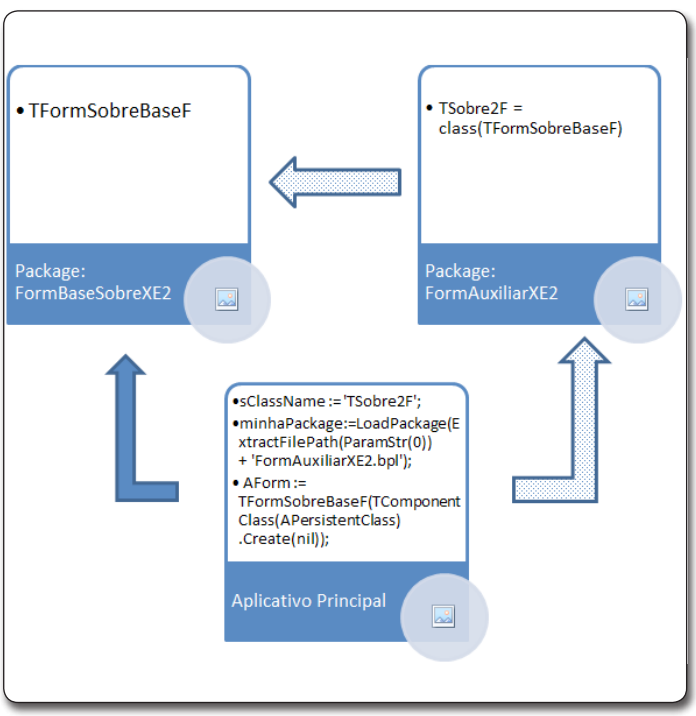


Figura 2. Relacionamento entre Packages

Estamos utilizando alguns conceitos da orientação a objetos aqui, como já mencionado utilizamos o polimorfismo, implementando em FormAuxiliarXE2 o que foi definido em FormBaseSobreXE2, agora no projeto principal vamos utilizar a herança para poder utilizar esses métodos que foram sobrecarregados. A **Listagem 8** mostra como isso foi feito.

Listagem 8. Acessando a BPL dinamicamente

```
01 procedure TPrincipalF.Button1Click(Sender: TObject);
02 var
03    minhaPackage: HModule;
04    versao: string;
05    APersistentClass: TPersistentClass;
06    sClassName: string;
07    AForm: TFormSobreBaseF;
08 begin
09  versao := '2.0';
10  sClassName := 'TSobre2F';
11  minhaPackage:=LoadPackage(ExtractFilePath(ParamStr(0))
12             + 'FormAuxiliarXE2.bpl');
13  if minhaPackage <> 0 then
14  try
15   APersistentClass := GetClass(sClassName);
16   if APersistentClass = nil then
17    raise Exception.Create('Classe não localizada!')
18  else
19  begin
20    AForm := TFormSobreBaseF(TComponentClass(APersistentClass)
21          .Create(nil));
22    AForm.SetaNumeroVersao(versao);
23    AForm.ShowModal;
24   if AForm.UsuarioMarcouAtualizao then
25    ShowMessage('Vamos atualizar ao iniciar')
26   else
27     ShowMessage('Não vamos atualizar ao iniciar');
28   FreeAndNil(AForm);
29  end;
30  finally
31   UnloadPackage(minhaPackage);
32  end;
33 end;
```

Nota do DevMan

Em orientação a objetos temos o conceito de herança, onde um tipo pode herdar as características de outro. Em código isso pode ser expresso da seguinte forma:

TForm1 = class(TForm)

Ao criar um novo formulário, é criada uma nova classe que herda de TForm, a classe base para formulários. Assim todo novo formulário herda suas características. Uma operação de Cast nada mais é do que fazer uma classe se passar por sua classe base. Temos duas formas de realização de cast, a direta e a indireta. O cast direto possui a seguinte sintaxe:

TForm(Form1)

Já o cast indireto é realizado através do operador as:

Form1 as TForm

Qual a principal diferença entre eles? O cast direto não é avaliado na compilação, ou seja, se tentarmos o acesso por uma classe base que não faz parte da hierarquia do objeto, o erro será percebido apenas em tempo de execução. Por isso essa forma de cast também é conhecida como unsafe (não seguro). O cast indireto também é conhecido como safe (seguro), pois é avaliado durante a compilação.

A principal diferença deste método com o anterior é que a aplicação principal conhece um tipo comum ao package dinâmico, a classe TFormSobreBaseF. Assim é possível realizar um cast direto (linha 20), pois sabemos também que o formulário a ser exibido herda dessa classe. Assim, temos acesso aos métodos definidos nela, que são implementados por seus descendentes.

Conclusão

Modularização é algo que deve ser bem pensado, estruturado. Justamente como nosso exemplo demonstrou, é preciso identificar o que será comum e isolar em um package que será utilizado por todos. Isso dá trabalho? Um pouco, mas os benefícios podem fazer valer, principalmente em sistemas com muitos formulários. Até a próxima!

Paulo Quicoli
pauloquicoli@gmail.com
Editor da revista ClubeDelphi e editor técnico .Net Magazine. Formado em processamento de dados pela FATEC-TQ. Atua como analista programador de sistemas e coordenador técnico da equipe de desenvolvimento .NET da Siplan Control-M (www.siplan-controlm.com.br) unidade Jaboticabal, e professor da FATEC, campus Taquaritinga.
Blog: http://quicoli.worpress.com

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

ER/Studio Business Architect

Modelando processos de negócios de forma simples e prática

O ER/Studio Business Architect, ou simplesmente ER/Studio BA é a ferramenta da Embarcadero que permite a representação gráfica do negócio (business), desde sua concepção até seus processos. Através de seus recursos é possível modelar processos por completo, abrangendo diferentes áreas do contexto. Para fluxos com envolvimento de dados, o modelo conceitual provido pela ferramenta contribui para a criação de um modelo de dados mais detalhado e eficaz. O ER/Studio BA é parte integrante da família de produtos do ER/Studio XE2, uma solução completa para arquitetura e modelagem de Softwares, negócios e dados. Sem deixar de mencionar, num passado não tão distante, o ER/Studio BA era conhecido como EA/Studio, no entanto, esta atribuição não é mais utilizada.

Em termos práticos, toda sua base é formada por um conjunto de três funcionalidades: modelagem, relatórios e análise gráfica. Por fazer uma abordagem direta sobre o negócio em si, é através de tais itens que a ferramenta irá fornecer informações para as diversas áreas da organização, cada qual com sua devida relevância. Citando exemplos, enquanto um gerente de projetos pode necessitar visualizar uma determinada lista das regras de negócios envolvidas, o desenvolvedor obtém um relatório gráfico contendo os dados utilizados por um processo.

Para a parte de modelagem de processos, a semântica utilizada pelo ER/Studio BA é toda baseada na BPMN (ver nota do DevMan), o que contribui para uma melhor percepção e um resultado rico em aspecto. No mais, a partir de sua utilização direta, a ferramenta acaba por atender a diferentes fins, variavelmente de acordo com o profissional atuante. Da mesma forma, citando um analista responsável pela modelagem de dados, por exemplo, este é capaz de criar modelos conceituais de alto nível. Consequentemente, atingindo um nível de aceitação suficiente, estes modelos poderão então servir de base para um modelo mais específico e detalhado, como um Modelo Lógico de dados.

Resumo DevMan

De que se trata o artigo:

Mantendo a intenção de apresentar ferramentas complementares ao Delphi no cotidiano de desenvolvimento, o presente artigo faz uma abordagem sobre o ER/Studio Business Architect. Da mesma forma que produtos como ER/Studio Data Architect e ER/Studio Software Architect, ele é membro da família ER/Studio XE2, uma solução completa e complementar ao desenvolvimento, provida pela Embarcadero. Diz-se complementar pelo fato das ferramentas abrangerem áreas como Data Modeling (modelagem de dados), Software Modeling (modelagem de Software) e Business Modeling (modelagem de negócios). Conforme será visto, este último é o foco do ER/Studio Business Architect.

Em que situação o tema é útil:

A modelagem de processos de negócio é uma forma padrão de representar os diversos processos intrínsecos a uma empresa. A partir da notação gráfica produzida, aperfeiçoa-se a atividade de análise sobre os processos, facilitando a busca por sua melhoria em termos de eficiência e qualidade. A modelagem conceitual é uma das segmentações da modelagem de dados que visa representar as características iniciais observadas no ambiente que está sendo analisado, sem determinar qualquer particularidade de implementação. Em vista disso, o ER/Studio Business Architect é a ferramenta da Embarcadero destinada à modelagem conceitual e de processos de negócios, conforme será visto no presente artigo.

ER/Studio Business Architect:

O Embarcadero ER/Studio XE2 apresenta-se como um agregado de produtos envolvidos com a modelagem de dados, aplicações e de negócios. Especificamente sobre este último, o produto em questão vem sob o nome de ER/Studio Business Architect, que visa facilitar as atividades de trabalho de um profissional arquiteto, provendo recursos suficientes para documentar processos de negócios e o envolvimento destes com dados. Por estar diretamente relacionada ao contexto de desenvolvimento de Software, a abordagem realizada pela ferramenta acaba por agregar valor não somente aos trabalhos realizados por tais profissionais. Prova disso é o próprio programador, visto que, nos dias atuais este tipo de profissional acaba por exercer várias funções, além de sua natural, que é a de lidar com o código da aplicação.

IDE

O IDE (Integrated Development Environment) do ER/Studio Business Architect é baseado em outro ambiente de desenvolvimento, o Eclipse, muito conhecido do mundo Java. Sua infraestrutura é então basicamente apoiada em cinco pilares – Workspace, Project, Model, Diagram e Object – que acabam por definir toda usabilidade e organização das atividades aplicadas ao ambiente.

Workspace

O Workspace é o espaço de trabalho do ambiente e de caráter único. Diz-se único, pelo fato do ER/Studio BA não trabalhar concorrentemente com vários Workspaces, permitindo que somente um esteja aberto por vez. Na cadeia estrutural, um Workspace é determinado como sendo Top Level, sendo o elemento de mais alto nível, servindo de container para outros itens que também serão manipulados pelo IDE, como projetos (Projects) e objetos de referência (Reference Objects).

Project

Contido num Workspace, um projeto é caracterizado por uma coleção de itens correlacionados, relevantes ao cliente e seu negócio como, por exemplo, modelos, documentos e regras de negócios. Estruturalmente falando, é possível o estabelecimento de uma relação pai-filho, onde um Project pode ser aninhado em outro existente. Cada Workspace pode conter N projetos.

Model

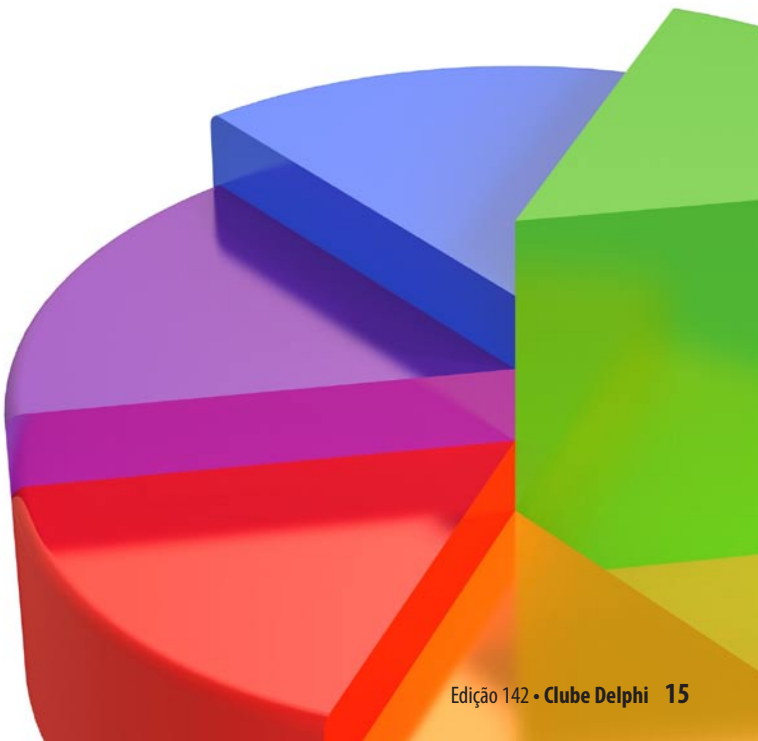
No ambiente do ER/Studio BA, o modelo é uma espécie de container que agrega elementos e objetos relacionados a uma determinada notação de modelagem, tal como a Business Process Modeling Notation (BPMN). Ele está diretamente ligado ao projeto, uma vez que, quando utilizado, este se torna o principal responsável por organizar os grupos de modelos. No entanto, um Workspace pode conter vários modelos, independente da presença de um elemento Project.

Object

São os elementos que compõem os modelos conceituais, bem como os processos de negócios elaborados no ER/Studio BA, tais como tarefas (Tasks), objetos de fluxo (Flow Objects) e entidades de negócio (Business Entities). De forma genérica, todos são tratados como sendo objetos.

Diagram

No IDE, um diagrama é o elemento utilizado para representar visualmente as operações de negócios, as chamadas Business Operations. Para isso, são então criados modelos gráficos que são expressos em três tipos principais de diagramas: Business Process Model, Conceptual Model e Impact Analysis. Este último pode ser considerado o menos usual, sendo um diagrama voltado à análise de impacto. Em suma, é onde estarão representadas graficamente as relações existentes entre os objetos utilizados. Enquanto isso, Business Process Model é o diagrama que representa visualmente o fluxo das atividades envolvidas e sua ordem de desempenho. A definição deste modelo envolve a utilização de um conjunto de elementos gráficos que acabam por representar algumas formas familiares aos utilizadores da ferramenta. Exemplo disso são as próprias atividades, que são graficamente representadas por um retângulo. A finalidade principal deste tipo de abordagem é poder modelar processos de negócios complexos através da criação de um modelo eficiente e de fácil entendimento. Conceptual Model define uma modelagem conceitual, que comumente é relacionada ao um modelo lógico de dados, podendo lidar inclusive com o fluxo de dados. Sendo assim, é voltada a uma determinada segmentação de profissionais, expressas por arquitetos de dados, de negócios e corporativos.



Nota do DevMan

Business Process Modeling Notation, ou simplesmente BPMN, pode ser definida como uma notação da metodologia de gerenciamento de processos de negócios. É mantida por uma organização chamada Object Management Group, responsável direta por sua desenvoltura e incremento. Tecnicamente falando, sua notação envolve a utilização de uma gama de ícones, todos apresentados num formato padrão para o esboço e desenho de processos, facilitando o entendimento por parte do usuário. Desde seu início este tipo de característica proporcionou uma considerável aceitação à BPMN, por parte de várias empresas do segmento.

Nota do DevMan

Eclipse é um ambiente de desenvolvimento integrado (IDE) de caráter open-source e totalmente desenvolvido em Java, sendo um dos mais utilizados no que tange o desenvolvimento de software com esta linguagem. Seu projeto foi concebido inicialmente pela empresa norte-americana IBM e, posteriormente, doado para a comunidade de Software Livre. Sua usabilidade não se restringe ao contexto Java, sendo empregada no desenvolvimento com outras linguagens, tais como PHP e C++, através de plug-ins específicos e até mesmo servindo de base para o surgimento de outras ferramentas.

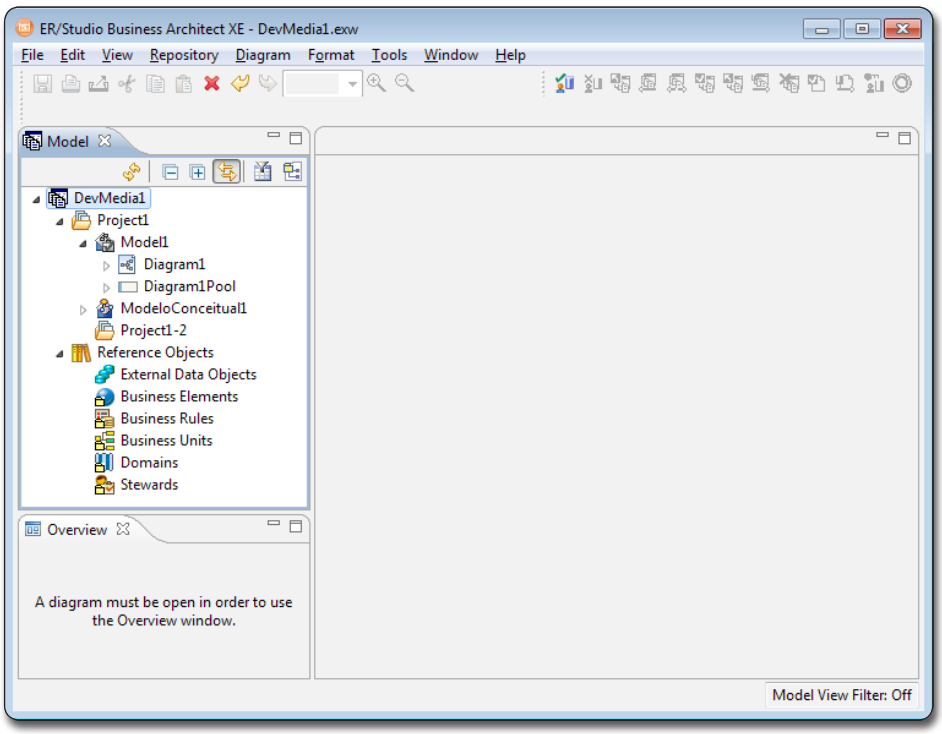


Figura 1. Overview do IDE

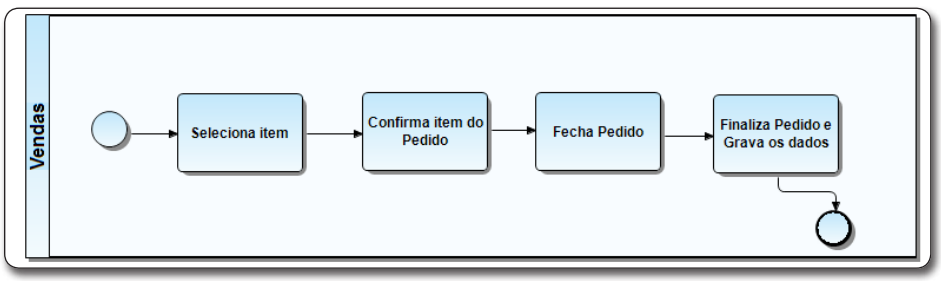


Figura 3. Trecho de processo mapeado em um Business Process Model

De forma ilustrativa, a **Figura 1** mostra os itens citados dispostos no IDE.

Model View – Visualizando e manipulando hierarquia de elementos

O IDE do ER/Studio Business Architect, assim como todos os outros produtos da família ER/Studio XE2, é composto de várias partes, cada qual com sua especificação. Dentre estas a Model View (**Figura 2**), que pode ser considerada a principal estrutura presente no ambiente. Isto se justifica pelo fato dela prover a exibição e gerenciamento da estrutura hierárquica de espaços de trabalho, projetos, modelos, diagramas e elementos de referência, que no IDE são tratados como Workspaces, Projects, Models, Diagrams e Reference Elements. Na árvore estrutural, cada item deste é representado por um ícone significativo que, com o passar do tempo, acaba por fazer o profissional associar determinado conceito ao ícone gráfico.

Na hierarquia citada, o modo com que os elementos são dispostos varia de acordo com o contexto ao qual estão inseridos. Basicamente, isto é determinado pelo tipo de modelagem que está sendo trabalhada, se Business Process Model ou Conceptual Model, através de seus diagramas. Neste quesito, a modelagem conceitual pode ser considerada a mais básica, tendo em vista que todo elemento criado é posto logo abaixo ao nó do modelo. Já para a modelagem de processos de negócio, em sua árvore todo e qualquer elemento é imediatamente

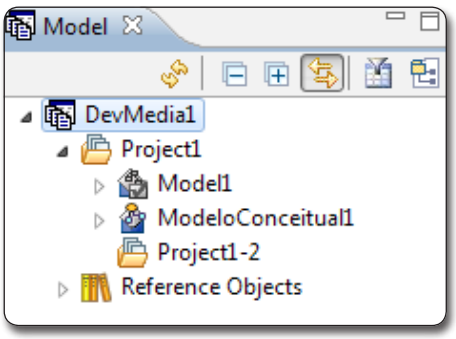


Figura 2. Model View

aninhado abaixo do nó de um processo (Process) ou de um Pool. Como exemplos mais comuns temos um evento cujo "proprietário" é um processo, da mesma forma que uma Lane torna-se uma ramificação de Pool. As **Figura 3** e **4** exibem esboços de estruturas relativas à Business Process Model e Conceptual Model sendo mostradas no IDE do ER/Studio BA. Elementos como Pool e Lane serão abordados na sequência do artigo.

Palletes – Auxiliando na criação de diagramas de modelagem

Outra estrutura de suma importância no IDE são as Palettes, ou paletas, que são as responsáveis por auxiliarem diretamente na criação dos diagramas de modelagem. Ao total, são sete as paletas disponíveis conforme mostra a **Figura 5**. São elas: Business Process Objects, Conceptual Objects, Reference Objects, External Data Objects e Drawing Shapes. Cada paleta desta dispõe então de um conjunto de objetos relacionados a um âmbito específico, que varia de acordo com o modelo ao qual se está trabalhando. Sendo assim, fica fácil presumir que ao se manipular um diagrama de processos de negócio, a paleta ativa será a Business Process Objects, da mesma forma que para um diagrama conceitual a paleta em questão será a Conceptual Objects.

Nestas paletas, cada objeto possui elementos adicionais associados, funcionando como uma espécie de categorização. Citando como exemplo a Business Process Objects, seus elementos são dispostos basicamente em treze categorias. Dentre as principais pode-se citar:

Start Events, Intermediate Events, End Events, Tasks, Gateways, Pools e Lanes. Apenas para citar, Start Events agrega os elementos relacionados ao início de um processo, tais como o Link, que é utilizado para se conectar o fim de um processo ao início de outro e Rule, que condiciona o início do processo a uma determinada regra. De forma semelhante, os elementos de Intermediate Events são os que demarcam acontecimentos durante o curso de um processo.

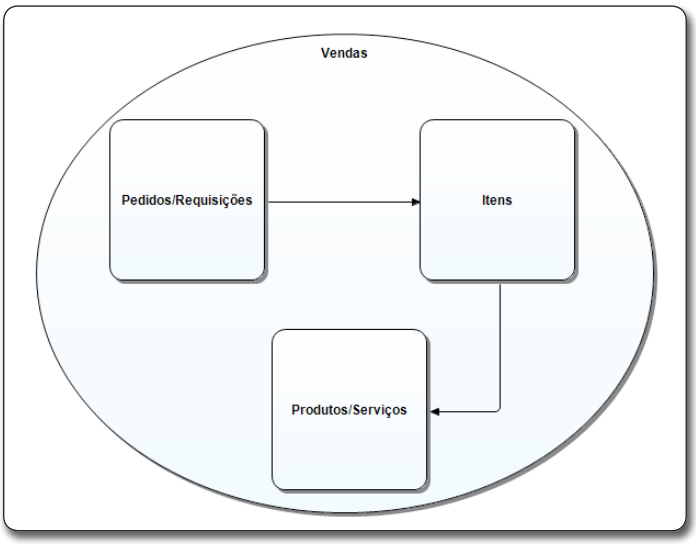


Figura 4. Exemplo de Modelo Conceitual (Conceptual Model)

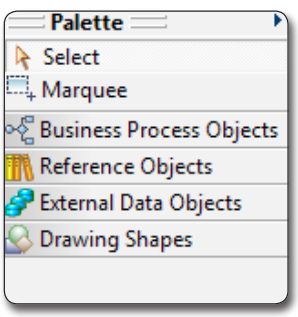


Figura 5. Palette

Iniciando a modelagem com o ER/Studio Business Architect

Olhando pelo lado prático, o ponto de partida para se começar a efetivamente modelar com o ER/Studio BA é provido através de sua caixa de diálogo inicial. Conforme exposto pela **Figura 6**, esta janela de boas-vindas do ambiente provê, ao mesmo tempo, opções para que se inicie toda e qualquer atividade a ser realizada no IDE. Partindo-se de uma utilização inicial comumente é então criado um novo Workspace, onde posteriormente serão agregados os eventuais projetos, modelos e diagramas. Não obstante, outras opções tradicionais como abrir o último Workspace utilizado ou qualquer outro existente também é possibilitado. Convém ainda obtê-lo diretamente do repositório (Repository), nos casos onde houver a integração com algum sistema de controle de versão. Neste caso em específico há o envolvimento de outro produto relacionado à família ER/Studio XE2, denominado ER/Studio Repository, que funciona aos moldes do Subversion para repositório e versionamento de arquivos.

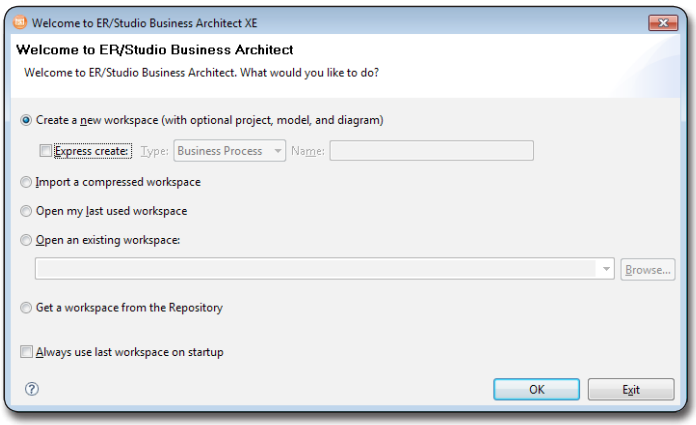


Figura 6. Tela de boas-vindas do IDE contendo opções para início prático

Criando Workspaces e elementos hierárquicos

Conceitualmente falando, um processo é caracterizado basicamente por uma coleção de objetos que se interligam a fluxos, sendo assim denominados como objetos de fluxo (Flow Objects). A ordem de sua execução é então representada por um diagrama, que expõe também sua sequência. No ER/Studio BA, ao se manipular modelos de processo de negócios (Business Process Models), o foco fica exclusivamente em torno do trabalho com processos. Sendo assim, levando em conta as especificações da BPMN, para este contexto de processos são introduzidos dois novos conceitos relacionados: Pool e Lane.

Pool é a representação dos principais participantes em um processo. Seu uso tem por objetivo separar as diferentes organizações, onde cada qual será representada por um Pool. Em termos estruturais um Pool pode conter apenas um processo, enquanto que um diagrama pode conter vários Pools. A comunicação entre Pools é realizada através de troca de mensagens inseridas em ocasiões onde há a presença de múltiplos processos de negócios que estão fisicamente separados. O termo Pool, cuja tradução livre para o português é piscina, é empregado neste cenário justamente por representar uma espécie de piscina real. Isto porque um Pool pode conter várias partições internas que, analogamente, representariam uma piscina e suas faixas. Cada sub-partição desta é então denominada Lane, sendo utilizada para organizar e categorizar as atividades internas a um Pool. Outra analogia muito utilizada para facilitar o entendimento da relação entre Pool e Lane é estabelecer que Pool representa uma organização, enquanto que Lane é um departamento dentro desta.

No ER/Studio BA, todos estes elementos citados ficam dispostos sob a hierarquia de um Workspace. Sendo assim, já no momento de sua criação abre-se a possibilidade para a definição de cada um dos itens (Project, Model, Diagram, Pool e Lane), conforme mostra a **Figura 7**.

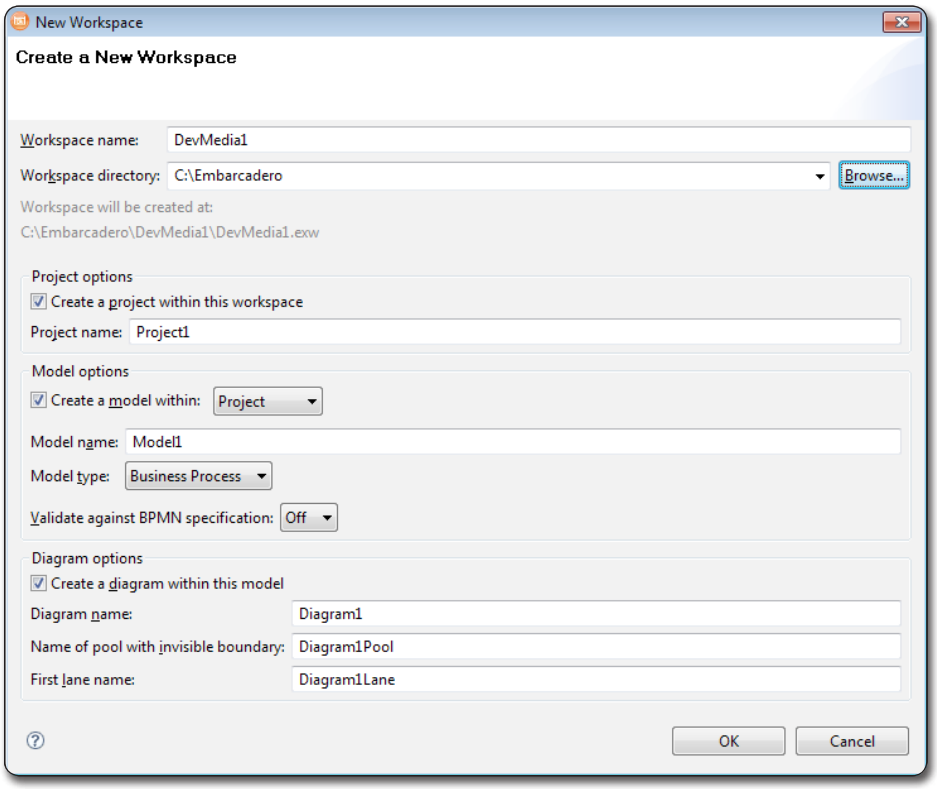


Figura 7. Definindo um Workspace e seus elementos

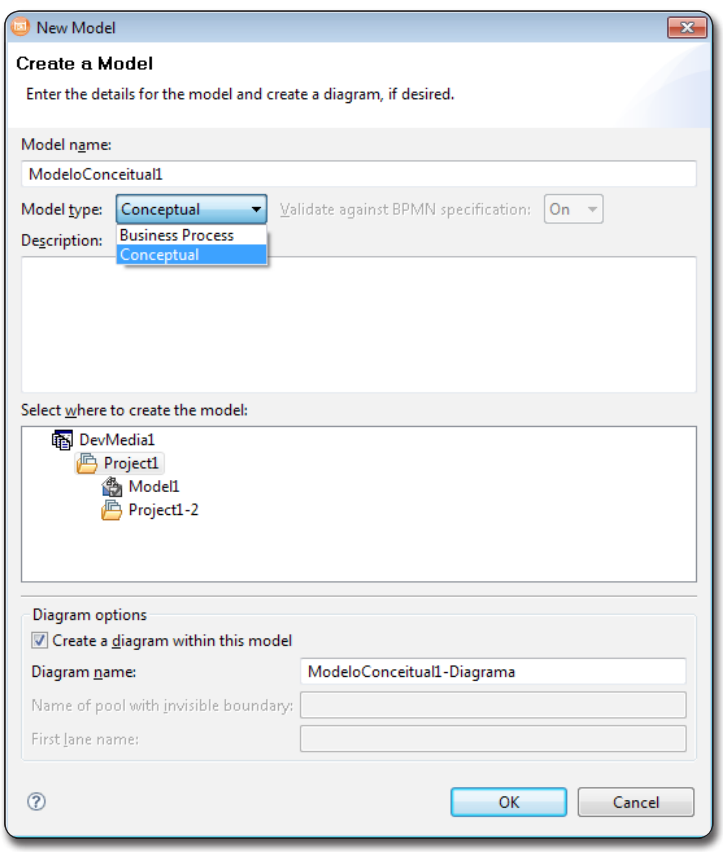


Figura 8. Criação de um novo modelo no ER/Studio BA

Modelo conceitual (Conceptual Model)

Modelo conceitual é o artefato utilizado para se modelar as informações relacionadas fundamentalmente sobre a parte de dados do negócio. Diz-se fundamentalmente pelo fato do modelo poder incluir ainda, além de dados, outros ativos de negócio, processo e fluxos. Sendo assim, este modelo é comumente definido na parte inaugural do projeto e seu escopo ficando a cargo de profissionais correlativos, tais como arquitetos (corporativos e de dados) e usuários de negócios (Business Users). Seu uso é justificado principalmente pelo fator organizacional, tendo em vista que o formato do modelo propicia o arranjo dos inúmeros objetos físicos que possam estar envolvidos nesta parte do projeto. Além disso, outro fator preponderante para a definição de um modelo conceitual é sua interoperabilidade entre a parte técnica e não técnica do negócio. Prova disso é que este tipo de modelo é simples o suficiente para ser devidamente assimilado por usuários não técnicos, ao mesmo tempo em que é o ponto de partida para exposição de detalhes mais técnicos, tais como os modelos lógico e físico de dados.

No ER/Studio Business Architect, o trabalho com modelos se dá pela manipulação de diagramas que podem ser essencialmente de dois tipos: Business Process e Conceptual. Sendo assim, durante a criação de um novo modelo é provida ao usuário a escolha do tipo (Model Type), bem como o estabelecimento de um diagrama (Diagram). Para um modelo conceitual as atividades serão exercidas sobre um diagrama conceitual (Conceptual Diagram), dessa forma, a **Figura 8** expõe a criação de um novo Modelo Conceitual onde podem ser vistos todos os detalhes citados anteriormente.

Modelo de processo de negócios (Business Process Model)

Em suma, um Business Process Model pode ser definido como um agregado de elementos gráficos que possibilitam expressar o fluxo de atividades relacionadas ao negócio, estabelecendo a ordem em que são desempenhadas. Assim como o Modelo Conceitual, o trabalho com

Modelo de Processo de Negócios no ER/Studio BA envolve o uso de diagramas. Aqui, cada diagrama é tratado como BPD, um acrônimo para Business Process Diagram, que expõe um fluxograma adaptado para modelar graficamente as operações de processos de negócios. Por este motivo um BPD pode ser considerado um diagrama simplificado, com a intenção de tornar seu desenvolvimento rápido e facilitado. Para isso, é então utilizada uma notação toda baseada em elementos gráficos familiares aos modeladores. Tais elementos tomam formas, tais como retângulos, losangos, diamantes e círculos, onde cada qual representa um item do processo. No IDE estes aspectos ficam dispostos em categorias, conforme mostrado anteriormente na **Figura 5**. Tudo isso resulta num mecanismo único para lidar com Modelos de Processo de Negócios, independente da complexidade relacionada.

No ER/Studio BA, no momento da criação de um novo Business Process Model, é possível determinar se o modelo deverá ser restrito às especificações da BPMN.

A premissa do ER/Studio BA - lidar com modelos

No ER/Studio Business Architect, trabalhar com modelos de processos de negócios significa manipular seus diagramas. De acordo com a BPMN, todo e qualquer diagrama relativo à Business Process deve necessariamente conter ao menos um Pool. Na ferramenta este conceito é devidamente empregado, sendo perceptível no instante da criação de um novo Diagrama, onde a indicação de um Pool é obrigatória. Citando a parte prática, o caminho natural para a definição de um novo Workspace contendo um digrama e Pool inicial é o mostrado anteriormente na **Figura 6**. Nestes casos, por padrão, os objetos inseridos em seu diagrama corrente são automaticamente colocados no primeiro Pool criado. Isto acontece pelo fato deste Pool inicial ser definido como **Pool with invisible boundary**, ou seja, uma espécie de piscina sem qualquer delimitação. Estas arestas só serão visíveis se por ventura este Pool for movido para outro diagrama.

No IDE do ER/Studio BA, os objetos a serem utilizados num diagrama de processos de negócios ficam disponíveis numa paleta exclusiva, tratada como Business Process Toolbar. Eles ficam agrupados de acordo com o contexto ao qual estão inseridos. Tomando como base o objeto Message Start Event, sobre ele é provido um menu drop-down contendo objetos adicionais relacionados, como Start Event, Link Start Event, Rule Start Event e Timer Start Event. Todos eles estão inseridos no contexto de eventos iniciais, os chamados Start Events, sendo conceitualmente muito próximos. Só para citar, enquanto um objeto Start Event demarca efetivamente o início de um processo, Message Start Event representa a mensagem que desencadeará o início do processo e Timer Start Event especifica o tempo ou ciclo que determinará o começo do processo.

O esboço de um diagrama de Business Process pode conter alguns dos objetos mais comuns ao cenário de processos de negócios. Basicamente há o envolvimento de itens como Start Event, Task, Sequence Flow e End Event. Task é o objeto com a forma de um retângulo e representa uma determinada atividade que é realizada dentro do processo de negócio. Já End Event, assim como Star Event, representa uma das extremidades da notação gráfica exposta, demarcando o fim do processo que está sendo tratado. Para se conectar uma atividade (Task) a outra, entra em cena um objeto especial denominado Sequence Flow ou fluxo de sequência, expresso por uma seta. Sua usabilidade vai além, determinando a conexão de eventos e de Gateways a Pools e Lanes, além de indicar a sequência de processos em uma organização. Um objeto Gateway tem como finalidade simbolizar decisões, fusões e junções no fluxo de um processo, estabelecendo um ponto de tomada de decisão.

Trabalhando com Objetos de Referência

Ao se definir um Workspace, por padrão, alguns objetos de referência são automaticamente criados. Isto pode ser notado na própria Model View através de um nó exclusivo denominado Reference Objects. Cada objeto deste assume um papel diferente no cenário do projeto, podendo ser de oito tipos diferentes:

- External Data Objects: tradicionalmente conhecidos como EDO's, objetos do tipo External Data Objects permitem o vínculo de objetos de dados externos a processos de negócios específicos. De forma mais abrangente, podem também ser visualizados em diagramas. Estes objetos de dados externos são então determinados por dois tipos distintos. O primeiro tipo, denominado Lógico, é representado pelas entidades e atributos, enquanto que o segundo, denominado Físico, é representado pelas tabelas e visões (Views).
- Business Element: basicamente, são elementos utilizados na documentação de lista de termos e definições. Ele pode estar atrelado a um atributo de negócio (Business Attribute) de um modelo conceitual, a fim de prover e impor regras e definições (Ex: padrões de nomenclatura);
- Business Rule: conforme sua nomeação indica, caracteriza uma regra de negócio relacionada às operações de negócios;
- Business Unit: uma unidade de negócios. Representa uma área organizacional interna ou externa à empresa;
- Domains: um domínio determina um modelo de atributo que será reutilizável por todo o contexto de modelo de dados. Exemplo disso é seu emprego nas colunas das diversas tabelas que um banco de dados pode possuir. Toda e qualquer alteração em um domínio é propagada às colunas que fazem referência a ele. Na prática, a definição de um Domain acontece aos moldes da definição de um atributo comum, em que se especificam determinadas propriedades, tais como nome, tipo de dado, estado e valor padrão;
- Stewards: um objeto do tipo Steward representa um elemento (pessoa ou organização) que esteja associado a um elemento do modelo. Em relação a Business Process Model, é possível se criar um Pool a partir de um objeto Steward;
- Link: pode ser considerado o objeto mais simples entre todos. É utilizado para estabelecer relações entre elementos, tais como objetos de dados, objetos de referência, Lanes e elementos individuais contidos em External Data Objects;
- Usage: este tipo cria um link rotulado como "usage", que indica um tipo de uso para operações CRUD sobre os dados de um banco de dados. Este indicativo é então relacionado a elementos presentes em diagramas, bem como objetos de dados e referência. Lembrando que CRUD é o acrônimo para Create, Read, Update e Delete, que representa as quatro operações básicas relacionada a dados: criar, ler, atualizar e excluir, respectivamente. No contexto do ER/Studio BA, cada operação desta é determinada como um atributo que indica seu uso. Como exemplo, Create é um atributo que indica uma usabilidade para a adição de novas entradas e registros.

Interoperabilidade: ER/Studio Business Architect e ER/Studio Data Architect

O ER/Studio XE2 é tido como uma família de produtos interrelacionados não por acaso. Isso se justifica pelo fato de suas ferramentas poderem se relacionar, de forma a interoperar arquivos e, consequentemente processos. É o caso da relação direta existente entre ER/Studio BA e ER/Studio DA. Na prática, durante o uso do ER/Studio Business Architect, objetos do tipo External Data Objects poderão ser "importados de" ou "exportados para" o ER/Studio Data Architect. Tais operações envolvem, essencialmente, entidades e tabelas.

O processo de importação de objetos EDOs se dá então a partir de arquivos do ER/Studio DA, por meio de Wizard disponível no menu exclusivo File>Import>ER/Studio Data Architect. Isso fará com que entidades e tabelas sejam importadas e automaticamente mapeadas para objetos de referência Table e Entity, respectivamente. A tela principal do assistente pode ser vista na **Figura 9**, onde pode ser notada a indicação para a seleção de um arquivo do ER/Studio DA.

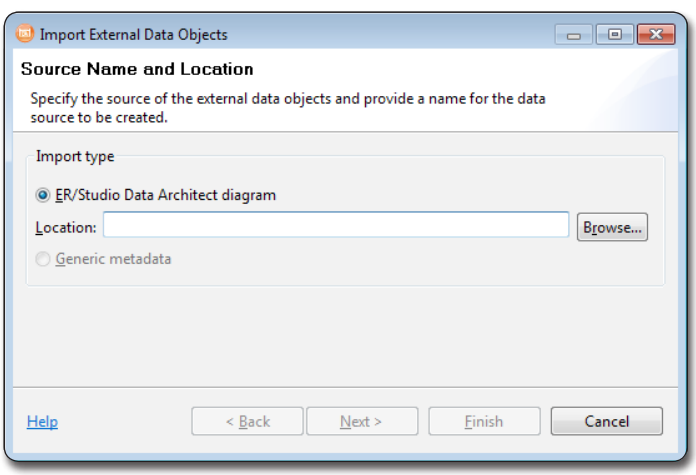


Figura 9. Wizard Import External Data Objects

Para o processo de exportação, a intenção é poder extrair External Data Objects do ER/Studio Business Architect para

o Data Architect. Sendo assim, elementos do tipo entidade (Entity) do primeiro, serão mapeados para um Modelo Lógico do segundo, da mesma forma que elementos do tipo tabela (Table) serão mapeados para um Modelo Físico.

Além desses, há ainda outra operação disponível relativa à interoperabilidade entre ER/Studio BA e ER/Studio DA, que é a de atualização de External Data Objects provindos do processo de importação. Para esta situação, novamente é introduzido um novo tipo de assistente, denominado Update External Data Objects Wizard (**Figura 10**). O resultado desta operação será a atualização dos objetos provindos de uma fonte de dados previamente importada. Na mesma medida, os objetos que não constam na estrutura atual serão importados e adicionados como novos.

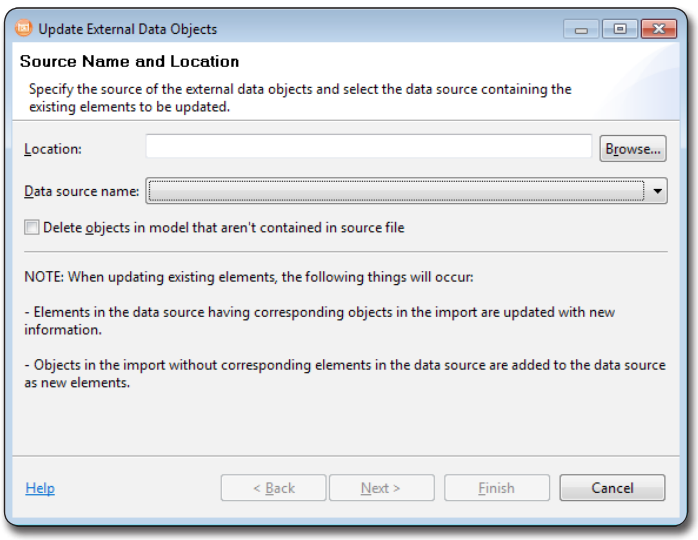


Figura 10. Update External Data Objects Wizard

Conclusão

A definição de um modelo conceitual é comprovadamente um artefato de grande importância durante a fase inicial de análise de um projeto. Visto sua concentração no mais alto nível de abstração do domínio do problema, não considerando qualquer tipo de particularidade relativa à determinada tecnologia de banco de dados. Conforme exposto pelo artigo, a manipulação deste tipo de modelo no ER/Studio Business Architect se torna facilitado em vista da gama de recursos disponíveis para sua construção e manutenção. Além disso, vale ressaltar a interoperabilidade existente entre as ferramentas pertencentes ao ER/Studio XE2, neste caso, o ER/Studio BA e o ER/Studio Data Architect. Prova disso é que um modelo conceitual construído no primeiro pode naturalmente ser aproveitado no segundo, servindo como base para concepção imediata do modelo lógico de dados.

Indo mais além, a ferramenta ainda possibilita a interação com o Microsoft Visio, provendo a importação de seus diagramas para Processos de Negócios e Modelos Conceituais. Na prática, para cada página de um modelo criado no Visio refletirá na criação de um diagrama no ER/Studio Business Architect. Portanto, a gama e possibilidades com a ferramenta são inúmeras. Agradeço a todos e até a próxima.

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ. Atua como Analista Programador Delphi há 6 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas.

Embarcadero ER/Studio Data Architect – Página oficial do produto
https://downloads.embarcadero.com/free/er_studio_data_architect

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

ORM com Aurelius

Aplicando persistência com o Framework da TMS

Resumo DevMan

De que se trata o artigo:

Este artigo aborda o emprego de ORM (Mapeamento Objeto Relacional), um recurso que possibilita a gravação ou conversão de objetos para registros em um banco de dados, utilizando o Framework da TMS Software, Aurelius, para o Delphi XE 2, demonstrando um simples exemplo de como empregá-lo no desenvolvimento de aplicações Orientadas a Objeto.

Em que situação o tema é útil:

Apresentar uma alternativa para o desenvolvedor que deseja elaborar aplicações que utilizem objetos para a manipulação das informações presentes em um banco de dados, obtendo inúmeras vantagens ao adotar este recurso em seus projetos.

ORM com Aurelius:

Atualmente, o conceito de Orientação a Objetos é extensamente aplicado no desenvolvimento de Software, pois apresenta inúmeras vantagens, além de que muitas linguagens de programação oferecem suporte ao mesmo. Contudo, ao integrar um sistema OO com um banco de dados podem surgir algumas questões, uma vez que funcionam de formas diferentes. É neste momento que a tecnologia ORM deve ser empregada, com o intuito de auxiliar o desenvolvedor na persistência dos seus objetos para o banco ou meio físico, aumentando assim a produtividade e tornando o sistema facilmente legível e portável. Este artigo tem o objetivo de demonstrar o recente Framework Aurelius para o emprego de ORM com o Delphi XE 2, apresentando um exemplo prático de como utilizá-lo no desenvolvimento de uma simples aplicação com o banco de dados SQLite e duas tabelas, mas que pode ser elaborada com qualquer outro banco suportado pelo Aurelius.

A Orientação a Objetos trouxe inúmeros benefícios para o desenvolvimento de Softwares cada vez mais robustos, flexíveis, escaláveis, entre outras características, provendo a arquitetura ideal para ser empregada na atualidade quando o assunto é planejamento de sistemas. Entretanto, durante o projeto de uma aplicação baseada em OO podem ocorrer alguns contratempos, uma vez que nem todos os recursos necessários para o funcionamento adequado do Software oferecem total suporte a este conceito ou trabalham de forma diferente, como é o caso de grande parte dos SGBDs da atualidade, que são baseados no popular e tradicional modelo relacional. Neste modelo, os dados são organizados através das relações, onde existem tabelas constituídas basicamente por campos e registros, empregando chave primária e estrangeira. Diante deste contexto, existe a necessidade de um recurso que permita que os objetos derivados das classes que representam as entidades possam ser "convertidos" e gravados em um banco de dados ou um meio físico, sendo tal recurso denominado ORM (Object-Relational Mapping – Mapeamento Objeto Relacional). Uma vantagem importante ao adotar ORM nos projetos é o aumento da produtividade, já que os códigos podem ser empregados diretamente em objetos, dispensando muitas vezes a utilização de complexas instruções SQL. Outra vantagem está relacionada à manutenção da aplicação que se torna mais fácil, uma vez que a lógica e as regras de negócio ficam mais evidentes quando objetos são empregados e a camada de banco de dados é abstraída. A portabilidade também é outro fator que deve ser levado em consideração, pois ao utilizar um bom Framework, o desenvolvedor pode migrar sua aplicação de banco de dados de uma forma simplificada, oferecendo ao seu cliente a possibilidade de escolher qual SGBD deseja adotar.

Um conceito muito importante para o emprego de ORM é a persistência de objetos. A persistência é o recurso responsável por possibilitar que as informações dos objetos presentes na memória volátil possam ser gravadas posteriormente em dispositivos de armazenamento permanente, utilizando um banco de dados, documentos XML (Extensible Markup Language), entre outros.

Quanto ao Delphi, após a implementação de recursos como Generics, que por sua vez permitem a utilização de tipos genéricos, listas fortemente tipadas, entre outros, surgiram diversas ferramentas que adicionaram novas possibilidades no desenvolvimento de aplicações. Dentre as tecnologias que nasceram recentemente, destacam-se os Frameworks que possibilitam o emprego de ORM, como é o caso do DORM (Delphi ORM) e em especial o Aurelius, que é uma ferramenta desenvolvida pela TMS e que oferece suporte completo a manipulação de dados, polimorfismo, consultas complexas, herança, dentre outros recursos. Contudo, este Framework é proprietário, mas possui um preço de licença acessível (195 €) e uma versão Trial que vem com todas as funcionalidades e pode ser facilmente obtida através do site da TMS (ver seção links), onde o desenvolvedor pode se familiarizar e efetuar todos os testes desejados com fins não comerciais.

Componente	Descrição
IDBConnection	Uma Interface que é utilizada para representar a conexão com um banco de dados através do Aurelius, oferecendo a base para as operações.
TDatabaseManager	Classe disponibilizada pelo Aurelius para a criação de tabelas e campos para o banco de dados com base nas classes elaboradas no Delphi.
TObjectManager	Classe responsável pela camada entre os objetos e o banco de dados, possibilitando a gravação, atualização, exclusão e carregamento das informações.

Tabela 1. Principais Componentes do Framework Aurelius

Banco de Dados	Componente	Adapter
dbExpress (DBX)	TSQLConnection	TDBExpressConnectionAdapter
Interbase (IBX)	TIBDatabase	TIBExpressConnectionAdapter
dbGo (ADO)	TADOConnection	TDbGoConnectionAdapter
AnyDac	TADDabase	TAnyDacConnectionAdapter
SQLite	Não necessita	TSQLiteNativeConnectionAdapter

Tabela 2. Principais Adapters para o componente IDBConnection

Banco de Dados	Unit
Firebird	Aurelius.Sql.Firebird
Interbase	Aurelius.Sql.Interbase
PostgreSQL	Aurelius.Sql.PostgreSQL
Oracle	Aurelius.Sql.Oracle
SQL Server	Aurelius.Sql.MSSQL

Tabela 3. Algumas Untis para o suporte ao SQL Dialect no framework Aurelius

O Framework Aurelius

O Aurelius é um importante recurso da atualidade para o emprego de ORM no Delphi XE 2 e também para outras versões. Ele permite a integração com diversos bancos de dados importantes, tais como o SQLite, Firebird, MySQL, Interbase, Oracle, SQL Server, PostgreSQL entre outros, suportando vários componentes de acesso como o dbExpress, IBX e ADO por exemplo, possuindo também acesso nativo para conexões com o SQLite. Este Framework também é multiplataforma, podendo ser empregado na elaboração de aplicações para Windows 32/64 bits e Mac OS, utilizando os recursos da VCL (Visual Component Library) para o Windows ou o novo ambiente de desenvolvimento do XE 2, o FireMonkey.

Com a utilização deste Framework, o desenvolvedor pode facilmente efetuar a gravação, atualização e carregamento das informações dos objetos com o banco de dados, criando consultas com uma API (Application Programming Interface) baseada em expressões, com suporte a agrupamentos, projeções, condicionais, dentre outros recursos. Além disto, ele permite o mapeamento automático das classes elaboradas no Delphi através de atributos, aumentando assim a produtividade no emprego de ORM. Este pode ser facilmente integrado ao Delphi XE 2, uma vez que basta obtê-lo através do site da empresa e efetuar sua instalação através de um executável, sendo também bem documentado, e, após a conclusão deste processo, basta adicioná-lo ao ambiente RAD através do menu Component>Install Packages.

Como todo componente de terceiros, o Aurelius também possui suas próprias características, classes e funções para que o mesmo possa ser utilizado. A **Tabela 1** apresenta os principais componentes do Framework Aurelius para implementação de ORM no projeto.

Quando uma conexão com um banco de dados é efetuada através do componente IDBConnection, existe a necessidade da criação de um Adapter (adaptador) para o componente de acesso utilizado, ou seja, se a conexão será efetuada com o SGBD Interbase, então o desenvolvedor deve utilizar o Adapter do mesmo, que por sua vez é o TIBExpressConnectionAdapter. A **Tabela 2** apresenta os principais Adapters que devem ser empregados em conjunto com o IDBConnection para possibilitar a conexão com o banco de dados.

Ao definir o banco de dados que será empregado e seu Adapter, também deve ser apontado qual é o SQL Dialect que será utilizado, uma vez que o Framework Aurelius vai trabalhar com comandos SQL para interagir com as informações. O SQL Dialect é uma derivação da linguagem SQL que por usa vez utiliza expressões de fácil legibilidade e que deve estar de acordo com o banco de dados que é aplicado. A **Tabela 3** apresenta algumas Units do Aurelius que devem ser incluídas no projeto para a utilização do SQL Dialect correto de acordo com o SGBD.

Uma breve introdução ao SQLite

Para dar início ao desenvolvimento ORM com o Aurelius, o primeiro passo é desenvolver um banco de dados, onde neste projeto foi adotado o SQLite, mas qualquer outro poderia ser empregado, uma vez que este Framework possibilita uma fácil por-

tabilidade. O SQLite é uma biblioteca de Software baseada em C multiplataforma no qual efetua a implementação de um banco de dados SQL que não necessita de um servidor nem de instalação e oferece suporte a transações, sendo seu código fonte disponibilizado livremente. Além disso, o SQLite é rápido e tem se tornado muito popular, uma vez que também está presente em dispositivos móveis. Entretanto, ele não oferece o recurso de integridade referencial.

Para criar o banco de dados deste exemplo, foi utilizado o Software Livre SQLite Administrator (ver seção links), que possui um ambiente em português, mas existem várias aplicações que podem ser utilizadas. A **Figura 1** apresenta a ferramenta para administração do banco de dados, que neste exemplo foi denominado "BASE.s3db", em conjunto com as duas tabelas e campos que foram criados para este exemplo.

Aurelius na prática

Com o banco de dados criado, basta iniciar um novo projeto de aplicação VCL (File>New>VCL Forms Application) para a utilização de ORM, lembrando a plataforma FireMonkey também pode ser utilizada. O primeiro passo é adicionar uma Unit (File>New>Unit) ao projeto com o intuito de armazenar as duas entidades que serão empregadas no sistema, fornecedor e produto, que neste exemplo foi denominada "UEntidades.pas". Como mencionado anteriormente, o Aurelius possui recursos que permitem o mapeamento automático das classes do Delphi, sendo o mesmo executado com a adição da Unit Aurelius.Mapping.Attributes e evidentemente alguns atributos na criação da classe. A **Tabela 4** apresenta os principais atributos que podem ser utilizados para o mapeamento das classes Delphi através do Aurelius.

Contudo, ao utilizar o atributo [Automapping] existem alguns fatores que devem ser levados em consideração pelo desenvolvedor. Ao declarar a classe, basta utilizar o prefixo "T" seguido pelo nome da tabela que o Framework se encarrega de efetuar a relação com a mesma, ou seja, se o nome da tabela é Cliente, então basta declarar TCliente. Também deve ser definida uma propriedade com a denominação de "FId" para que o mapeamento automático utilize-a como identificador, caso contrário o mesmo deve ser declarado manualmente. Evidentemente, cada propriedade da classe deve ter o mesmo nome de cada campo da tabela, onde o prefixo "F", já utilizado para a definição de uma propriedade nas classes do Delphi, também pode ser empregado, em outras palavras, se o nome do campo é RazaoSocial então basta declarar a propriedade com o nome de FRazaoSocial. Entretanto, o mapeamento automático não é um recurso do tipo tudo ou nada, em outras palavras, o desenvolvedor pode utilizar o atributo [Automapping] em conjunto com outros atributos manuais se quiser especificar propriedades diferentes interligadas ou não a tabela.

Sendo assim, o próximo passo é criar as classes referentes às entidades de acordo com as tabelas do banco de dados. A **Listagem 1** apresenta a classe TFornecedor e a TProduto que foram criadas para este projeto.

Neste trecho, são definidos os atributos [Entidy] e [Automapping] para indicar ao Aurelius que as classes são entidades e que o mapeamento automático deve ser empregado. Também foram definidos os campos de identificação "FId", necessários para cada classe, bem como cada campo presente nas tabelas criadas anteriormente com a utilização do prefixo "F". Para concluir em ambas as classes, foram criadas propriedades públicas para permitir a leitura (read) e gravação (write) das propriedades privadas e encapsuladas de cada classe.

Definidas as Entidades, o próximo passo é criar uma nova Unit, que neste projeto foi denominada "UConexao.pas", com o intuito de definir uma classe que será responsável pelo gerenciamento da conexão com o banco de dados e prover a base para que os objetos possam ser gravados, atualizados e carregados, que por sua vez recebeu a denominação de TGerenciadorConexao. Esta classe será do tipo sealed (selada). Este tipo sempre deve ser empregado quando determinada classe não deve ser herdada. Para efetuar a conexão com o banco será utilizado o componente IDBConnection comentado anteriormente, portanto, a Unit Aurelius.Drivers.Interfaces deve ser incluída na seção uses, assim como a Aurelius.Drivers.SQLite, uma vez que o banco utilizado será o SQLite. A **Listagem 2** apresenta o código inicial da Unit UConexao.pas e a declaração da classe TGerenciadorConexao.

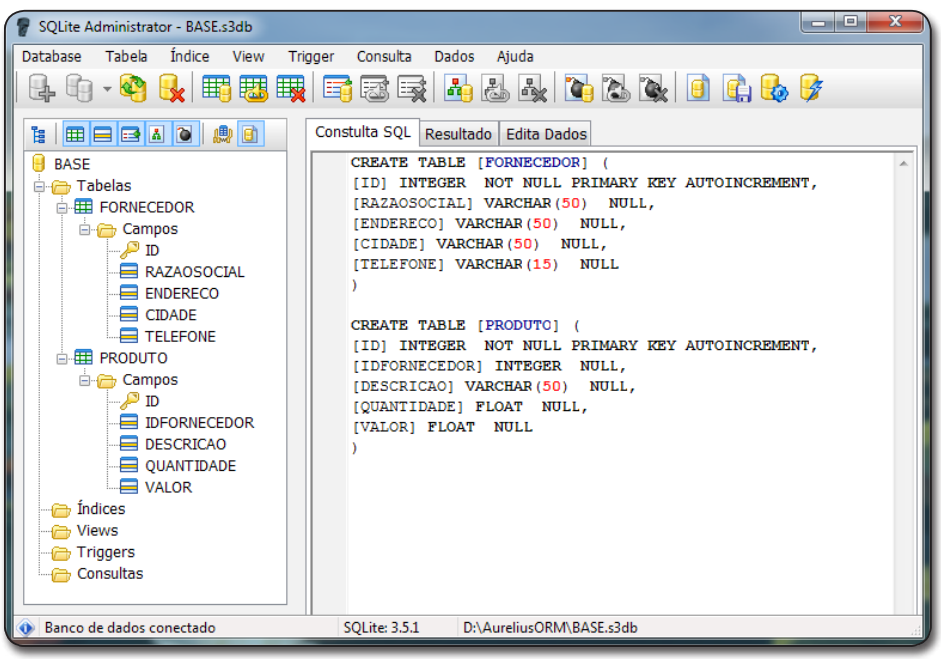


Figura 1. Interface Gráfica da ferramenta SQLite Administrator com o banco de dados utilizado neste exemplo.

Atributo	Descrição	Exemplo de Uso
[Entity]	Declarado antes da criação da classe para indicar uma entidade que por sua vez pode ser persistida.	[Entity] TCliente = class ...
[Automapping]	Declarado antes da elaboração da classe para efetuar o mapeamento automático da classe através do Aurelius.	[Entity] [Automapping] TCliente = class ...
[Table('Nome Tabela')]	Também deve ser declarado antes da criação da classe e pode ser utilizado no mapeamento manual para indicar que a classe esta relacionada à determinada tabela do banco de dados para que a gravação dos objetos possa ser efetuada.	… [Table('Cliente')] TCliente = class private ...
[Colum('Nome Coluna')]	Utilizada para o mapeamento manual de uma coluna da tabela para uma propriedade da classe, devendo ser definida antes da declaração da mesma.	... private [Column('Descricao')] FDescricao: string; ...
[Id('Campo Id'), Opções]	Deve ser utilizado para definir o identificador da classe de forma manual, o campo de chave primária no banco de dados. Em conjunto com a declaração do mesmo, devem ser definidas as opções, tais como auto-incremento.	... [Id('FId', TIdGenerator.IdentityOrSequence)] FId: Integer; ...
[Sequence('Nome Seq')]	Define um Generator ou Sequence que deve ser utilizado em conjunto com o campo Id para incremento automático.	... [Sequence('Cliente_Seq')] [Id('FId', ...
[Association(Opções)]	Utilizado para indicar relacionamentos entre as entidades (outras classes), tais como muitos-para-um, devendo ser empregada em conjunto com o atributo [JoinColumn].	... [Association([], CascadeTypeAll)] ...
[JoinColumn('Nome Coluna'), Opções]	Utilizado para especificar qual coluna da tabela deve ser utilizada para chave estrangeira.	... [JoinColumn('Id', [])] FIdX: Integer; ...
[Transient]	Utilizado para declarar uma propriedade da classe que não deverá ser persistida, também deve ser aplicado antes da definição do campo.	... [Transient] FNaoMapear: string; ...

Tabela 4. Principais Atributos empregados no Aurelius para o mapeamento

Listagem 1. Código das classes criadas na Unit Entidades.pas

```
uses
  Aurelius.Mapping.Attributes;

type
  [Entity]
  [Automapping]
  TFornecedor = class
    private
      FId: Integer;
      FRazaosocial: string;
      FEndereco: string;
      FCidade: string;
      FTelefone: string;
    public
      property Id: Integer read FId;
      property Razaosocial: string read FRazaosocial write FRazaosocial;
      property Endereco: string read FEndereco write FEndereco;
      property Cidade: string read FCidade write FCidade;
      property Telefone: string read FTelefone write FTelefone;
  end;

  [Entity]
  [Automapping]
  TProduto = class
    private
      FId: Integer;
      FIdfornecedor: Integer;
      FDescricao: string;
      FQuantidade: Double;
      FValor: Double;
    public
      property Id: Integer read FId;
      property Idfornecedor: Integer read FIdfornecedor write FIdfornecedor;
      property Descricao: string read FDescricao write FDescricao;
      property Quantidade: Double read FQuantidade write FQuantidade;
      property Valor: Double read FValor write FValor;
  end;
```

Listagem 2. Início do código da Unit UConexao.pas

```
uses
  Generics.Collections, Classes,
  System.SysUtils,
  Aurelius.Commands.Listeners,
  Aurelius.Drivers.Interfaces,
  Aurelius.Engine.AbstractManager,
  Aurelius.Engine.ObjectManager,
  Aurelius.Engine.DatabaseManager,
  Aurelius.Drivers.SQLite,
  Aurelius.Sql.SQLite;

type
  TGerenciadorConexao = class sealed
  private
    class var FInstancia: TGerenciadorConexao;
    FDBConnection: IDBConnection;
    procedure Destruir;
    function GetConexao: IDBConnection;
    function CriarConexao: IDBConnection;
  public
    class function GetInstancia: TGerenciadorConexao;
    function Conectado: Boolean;
    property Conexao: IDBConnection read GetConexao;
  end;

const
  EnderecoBanco = 'D:\AureliusORM\BASE.s3db';
```

Neste trecho de código são definidas todas as Units necessárias do próprio Delphi e do Framework Aurelius para a criação do gerenciador de conexão. A classe TGerenciadorConexao é definida como selada (sealed) para não ser herdada. É criada uma variável de classe (FInstancia) para permitir o acesso a instância atual da classe, através da futura função GetInstancia. Também é criada uma propriedade do tipo IDBConnection para armazenar a conexão. Posteriormente são definidas as funções GetConexao para obter a conexão atual, CriarConexao para efetuar uma nova conexão ao banco, e Conectado para acessar o estado da conexão, bem como a propriedade Conexao, quer permite o acesso ao IDBConnection. Para concluir, também foi definida uma constante com o intuito de armazenar o endereço do banco SQLite que será utilizado.

Após a declaração da classe, devem ser criados os procedimentos e funções que foram definidos. A **Listagem 3** apresenta a função CriarConexao que pertence a classe TGerenciadorConexao.

Listagem 3. Função CriarConexao

```
function TGerenciadorConexao.CriarConexao: IDBConnection;
begin
  if FDBConnection <> nil then
    Exit(FDBConnection);
  if FDBConnection = nil then
    FDBConnection := TSQLiteNativeConnectionAdapter.Create(EnderecoBanco);
  Result := FDBConnection;
end;
```

Nesta função primeiramente é verificado se o membro da classe (FDBConnection) é diferente de nulo, implicando que o mesmo já foi criado, sendo assim é efetuada uma saída da execução atual (Exit), uma vez que esta função pode ser chamada várias vezes durante a execução do programa. Caso seja a primeira utilização do FDBConnection ou mesmo ainda não tenha sido criado, então a conexão é concebida com o Adpater utilizado para o SQLite, TSQLiteNativeConnection.Create(), tendo como parâmetro o endereço do banco de dados que será utilizado, definido na constante EnderecoBanco para este exemplo, mas que poderia ser substituída por um valor presente em um arquivo INI. Como o banco utilizado é o SQLite, o Aurelius oferece uma conexão nativa ao mesmo. Para concluir, o resultado da função é o componente de conexão.

A próxima etapa é criar a função GetConexao, que efetuará uma chamada a função CriarConexao. A **Listagem 4** demonstra o código da função GetConexao.

Nota do DevMan

Arquivos do tipo INI são documentos baseados em texto que podem ser utilizados para armazenar diversas configurações das aplicações e normalmente são carregados no momento em que o Software é iniciado. São constituídos basicamente por seções, parâmetros e valores. O Delphi permite a fácil manipulação de arquivos INI com as funções da Unit IniFiles.

Listagem 4. Função GetConexao

```
function TGerenciadorConexao.GetConexao: IDBConnection;
begin
  Result := CriarConexao;
  if not Result.IsConnected then
    Result.Connect;
  if Result = nil then
    raise Exception.Create('Houve um erro ao criar a conexão!');
end;
```

Neste trecho, a função CriarConexao é chamada e caso o objeto de conexão de retorno (IDBConnection) ainda não estiver conectado, é efetuada a conexão. Para concluir, se não houve resultado é gerada uma exceção.

A próxima função é a GetInstancia, cujo objetivo é obter a instância atual da classe que foi criada para futuras conexões. A **Listagem 5** apresenta a função GetInstancia que pertence a classe TGerenciadorConexao.

Listagem 5. Função GetInstancia

```
class function TGerenciadorConexao.GetInstancia: TGerenciadorConexao;
begin
  if FInstancia = nil then
    FInstancia := TGerenciadorConexao.Create;
  Result := FInstancia;
end;
```

Neste trecho de código é possível notar que é efetuada uma verificação na variável definida anteriormente (FInstancia), com o intuito de testar se já existe uma instância da classe criada, e caso ela seja nula, então a classe TGerenciadorConexao é instanciada e atribuída a mesma variável, que por sua vez é retornada pela função.

O último código que deve ser inserido para esta Unit será utilizado para liberar os recursos da memória, portanto foi adicionado na seção finalization. Esta seção é executada no momento em que a Aplicação é terminada, lembrando que ela é opcional, portanto não é criada automaticamente pelo Delphi. Também deve ser definida a seção inicialization. A **Listagem 6** apresenta o código da seção finalization da Unit UConexao.pas.

Listagem 6. Seção finalization da Unit UConexao.pas

```
finalization
  if TGerenciadorConexao.FDBConnection <> nil then
  begin
    TGerenciadorConexao.FDBConnection.Disconnect;
    TGerenciadorConexao.FDBConnection := nil;
  end;
  if TGerenciadorConexao.FInstancia <> nil then
  begin
    TGerenciadorConexao.FInstancia.Free;
    TGerenciadorConexao.FInstancia := nil;
  end;
```

Neste trecho é efetuada uma verificação se o elemento FDBConnection é diferente de nulo, e caso seja, o mesmo é utilizado para desconectar do banco de dados e recebe o valor nulo, e posteriormente, a variável FInstancia também é verificada para que sua liberação da memória seja efetuada.

Criadas as Units contendo as classes e funções necessárias para o projeto, o próximo passo é empregá-las na aplicação. Para tanto, no formulário principal foram adicionados alguns componentes. A **Figura 2** apresenta o visual desta aplicação para o teste do projeto.

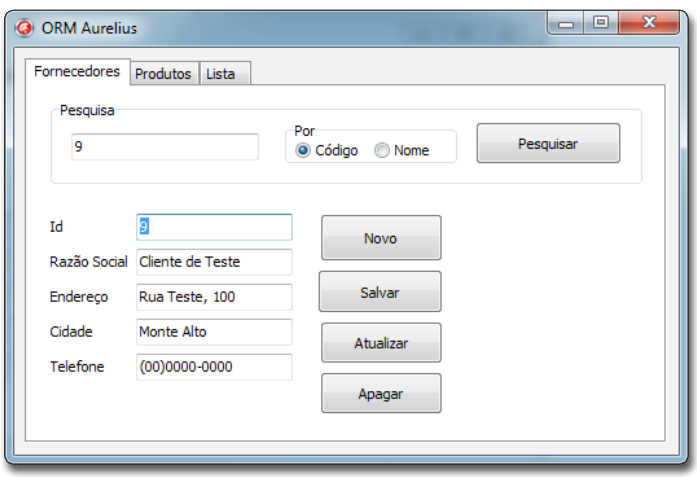


Figura 2. Visual da aplicação evidenciando o cadastro de fornecedores.

Sendo assim, as Units UConexao.pas e Uentidades.pas devem ser adicionadas à seção Uses do formulário principal. A primeira implementação para este exemplo foi efetuada para a classe TFornecedor. Sendo assim, na seção public do formulário principal foram criados dois objetos do tipo TObjectManager, o gerenciador de objetos do Framework Aurelius, que neste exemplo foram denominados "GerenciadorFornecedores", e "GerenciadorProdutos", e que posteriormente serão empregados. Também foram criados outros dois objetos, um do tipo TFornecedor, denominado FFornecedorAtual, e outro do tipo TProduto, denominado FProdutoAtual, com o intuito de auxiliar nas futuras operações que serão efetuadas. Também foi elaborada uma simples função neste formulário para obter a conexão atual. A **Listagem 7** apresenta o código da função GetConexaoAtual, bem como a inicialização dos objetos nos eventos OnCreate e OnDestroy do formulário.

Listagem 7. Função GetConexaoAtual e eventos OnCreate e OnDestroy

```
function TFrPrincipal.GetConexaoAtual: IDBConnection;
begin
  Result := TGerenciadorConexao.GetInstancia.Conexao;
end;

procedure TFrPrincipal.FormCreate(Sender: TObject);
begin
  GerenciadorFornecedores := TObjectManager.Create(GetConexaoAtual);
  GerenciadorProdutos := TObjectManager.Create(GetConexaoAtual);
end;

procedure TFrPrincipal.FormDestroy(Sender: TObject);
begin
  GerenciadorFornecedores.Free;
  GerenciadorProdutos.Free;
end;
```

A função GetConexaoAtual simplesmente invoca a classe TGerenciadorConexao para obter a conexão atual ou criar uma nova, sendo assim, no momento em que o formulário é criado (OnCreate), os dois objetos de gerenciamento deste exemplo também são criados utilizando esta conexão, e consequentemente na destruição do formulário (OnDestroy) os objetos são liberados da memória.

O próximo passo é iniciar o desenvolvimento dos códigos para cada operação de dados. A **Listagem 8** apresenta o código que foi atribuído ao botão btNovo.

Listagem 8. Código do evento OnClick do botão btNovo

```
procedure TFrPrincipal.btNovoClick(Sender: TObject);
var
  i: Integer;
begin
  for i := ComponentCount -1 downto 0 do
    if Components[i].ClassType = TEdit then
      TEdit(Components[i]).Clear;
  FFornecedorAtual := nil;
end;
```

O trecho de código da **Listagem 8** é uma simples maneira de automatizar a limpeza de valores preenchidos em todos os objetos instanciados da classe TEdit presentes no formulário, sendo muito útil em casos onde o número de componentes instanciados desta classe é elevado. De acordo com tal trecho, é efetuado um laço do tipo for com o intuito de percorrer todos os componentes do formulário, através de ComponentCount em conjunto com a variável de controle "i", e verificar a classe dos mesmos, comparando se o componente atual (Components[i]) é uma instância da classe TEdit, e caso verdadeiro, o texto digitado no componente é apagado com a utilização do método Clear. Para concluir, o objeto que é utilizado para armazenar o fornecedor atual é definido como nulo.

O próximo botão a ser implementado é o responsável pela gravação e denominado btSalvar. A **Listagem 9** apresenta seu código.

Listagem 9. Código do evento OnClick() do botão btSalvar

```
procedure TFrPrincipal.btSalvarFornecedorClick(Sender: TObject);
var
  Fornecedor: TFornecedor;
begin
  Fornecedor := TFornecedor.Create;
  Fornecedor.RazaoSocial := edRazaoSocial.Text;
  Fornecedor.Endereco := edEndereco.Text;
  Fornecedor.Cidade := edCidade.Text;
  Fornecedor.Telefone := edTelefone.Text;
  try
    GerenciadorFornecedores.Save(Fornecedor);
    GerenciadorFornecedores.Flush;
    edId.Text := IntToStr(Fornecedor.Id);
    FFornecedorAtual := Fornecedor;
    ShowMessage('Fornecedor gravado com êxito!');
  except
    ShowMessage('Houve um erro ao gravar!');
  end;
end;
```

Em um primeiro momento, é criada uma variável do tipo TFornecedor que logo em seguida recebe uma instância desta classe através de Create. Na sequência, todas as propriedades da classe são definidas de acordo com os valores dos TEdits, com exceção do Id, que por sua vez será gerado automaticamente. Após a definição das propriedades, o objeto GerenciadorFornecedores é chamado para gravar o objeto atual através do método Save e eftuar um Commit no banco de dados através do método Flush. Para concluir, o Id obtido para a Pessoa gravada é exibido no respectivo TEdit e o objeto FFornecedorAtual recebe a classe fornecedor, exibindo uma mensagem de gravação de acordo com a situação.

Nota do DevMan

Em banco de dados, um dos recursos mais utilizados é a transação, que por sua vez tem um início, uma sequência de comandos e um fim, onde o comando Commit é chamado para confirmar e concluir a mesma se a execução foi efetuada sem erros. Lembrando que em contrapartida, também há o comando Rollback cujo objetivo é anular tudo o que foi efetuado desde o início da transação caso ocorra algum erro.

A próxima etapa é implementar a pesquisa de um fornecedor para o carregamento de suas informações e também possibilitar a edição do mesmo. A **Listagem 10** apresenta o código que foi utilizado no botão btPesquisar.

Listagem 10. Código do evento OnClick do botão btPesquisar

```
procedure TFrPrincipal.btPesquisaClick(Sender: TObject);
begin
  if Trim(edPesquisa.Text) <> '' then
  begin
    try
      GerenciadorFornecedores.OwnsObjects := False;
      case rgCampo.ItemIndex of
        0: FFornecedorAtual := GerenciadorFornecedores.Find<TFornecedor>
       (StrToInt(edPesquisa.Text));
        1: FFornecedorAtual := GerenciadorFornecedores
          CreateCriteria<TFornecedor>
          .Where(TLinq.Like('Razaosocial', edPesquisa.Text)).UniqueResult;
      end;
      if FFornecedorAtual <> nil then
      begin
        edId.Text := IntToStr(FFornecedorAtual.Id);
        edRazaoSocial.Text := FFornecedorAtual.Razaosocial;
        edEndereco.Text := FFornecedorAtual.Endereco;
        edCidade.Text := FFornecedorAtual.Cidade;
        edTelefone.Text := FFornecedorAtual.Telefone;
      end
      else
        ShowMessage('O Fornecedor não foi encontrado!');
    except
      ShowMessage('Houve um erro!');
    end;
  end;
end;
```

Após a validação do preenchimento do campo de pesquisa com o auxílio da função Trim, utilizada para a remoção de espaços indesejados, é definida a propriedade OwnsObjects como falsa. Se o valor desta propriedade é True, todos os objetos gerenciados são destruídos com o gerenciador no momento em que o mesmo for destruído, caso contrário, os objetos continuam na memória. De acordo com a opção que foi selecionada pelo usuário, o objeto GerenciadorFornecedores efetuará a pesquisa e o objeto encontrado (se houver) é armazenado no objeto FFornecedorAtual. Se a mesma for efetuada através do código, basta utilizar o método Find, cujo objetivo é efetuar uma pesquisa para encontrar o objeto, seguido do tipo da entidade e o valor do campo, que neste caso é o Id digitado no TEdit. Caso o usuário selecione a pesquisa através do nome, então é utilizado um código com a intenção de efetuar uma pesquisa equivalente ao uso do comando SQL LIKE. No Aurelius, uma Query ou consulta pode ser realizada através de um objeto TObjectManager e do método CreateCriteria, que é baseado em critérios que devem ser informados para que a busca possa ser realizada, como a utilização da cláusula WHERE. Também é utilizado o componente TLinq pertencente ao Framework (Aurelius.Criteria.Linq), possui diversos comandos para efetuar comparações de forma semelhante as Expressões SQL. Para concluir, se o objeto FFornecedorAtual for diferente de nulo,

então as informações do mesmo são carregadas nos controles, caso contrário é exibida a mensagem informando que o fornecedor não foi encontrado.

A próxima etapa deste projeto é implementar o código do botão btAtualizar, com o intuito de permitir a edição de um fornecedor e a sua gravação. A **Listagem 11** apresenta o código que foi definido no evento OnClick do botão btAtualizar.

Neste trecho de código, o objeto de auxílio FFornecedorAtual, utilizado para carregar um fornecedor que foi encontrado pela função de pesquisa, recebe as alterações efetuadas nos componentes do tipo TEdit, evidentemente se o mesmo não estiver nulo. Terminado este processo, o objeto GerenciadorFornecedores é empregado para efetuar a atualização através do método Update e em seguida, o mesmo processo de Commit realizado na listagem anterior com o método Flush. Para concluir, a mensagem de gravação efetuada é exibida.

Para finalizar esta etapa, resta implementar o código responsável por deletar um objeto e evidentemente seu registro no banco de dados, utilizando o botão btApagar. A **Listagem 12** apresenta o código do evento OnClick do btApagar.

De acordo com o código, se o objeto FFornecedorAtual não for nulo então o GerenciadorFornecedores deleta o fornecedor selecionado através do método Remove, efetivando a operação com o Flush. Para concluir, efetua um clique no botão btNovo, que por sua vez limpa os campos, e exibe a mensagem de exclusão efetuada.

Concluído o gerenciamento das operações para os fornecedores, o próximo passo é implementar as operações para os produtos. Um ponto importante com relação ao cadastro de produtos é quanto

Listagem 11. Código do evento OnClick do botão btAtualizar

```
procedure TFrPrincipal.btAtualizarClick(Sender: TObject);
begin
  if FFornecedorAtual <> nil then
  begin
    FFornecedorAtual.RazaoSocial := edRazaoSocial.Text;
    FFornecedorAtual.Endereco := edEndereco.Text;
    FFornecedorAtual.Cidade := edCidade.Text;
    FFornecedorAtual.Telefone := edTelefone.Text;
    try
      GerenciadorFornecedores.OwnsObjects := False;
      GerenciadorFornecedores.Update(FFornecedorAtual);
      GerenciadorFornecedores.Flush;
      ShowMessage('Fornecedor gravado com êxito!');
    except
      ShowMessage('Houve um erro ao gravar!');
    end;
  end
  else
    ShowMessage('Selecione um fornecedor para alterar!');
end;
```

Listagem 12. Código do evento OnClick do botão btApagar

```
procedure TFrPrincipal.btApagarClick(Sender: TObject);
begin
  if FFornecedorAtual <> nil then
  begin
    GerenciadorFornecedores.Remove(FFornecedorAtual);
    GerenciadorFornecedores.Flush;
    btNovo.Click;
    ShowMessage('Fornecedor excluído com êxito!');
  end;
end;
```

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre ouutros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DevMedia

à implementação de uma rotina que solicite que um fornecedor esteja selecionado para que o produto possa ser inserido e gravado, uma vez que a tabela de produtos deve receber o id do fornecedor como chave estrangeira.

O desenvolvimento do cadastro de produtos fica como uma tarefa extra para a familiarização com o Framework Aurelius, uma vez que os códigos que serão utilizados para as operações com os mesmos são praticamente idênticos aos que foram empregados para os fornecedores, entretanto o código completo será encontrado através do download do exemplo.

Como neste projeto foi adotado que um fornecedor pode vender vários produtos, será implementada uma busca para todos os produtos de um mesmo fornecedor. Para a realização da pesquisa, foram adicionados além de uma nova página no PageControl, um StringGrid para carregar todos os produtos cadastrados para o fornecedor, além de um botão que efetuará a pesquisa. A **Listagem 13** apresenta o código do btListar para exibir todos os produtos do fornecedor.

Listagem 13. Código do evento OnClick do botão btListar

```
procedure TFrPrincipal.btListarClick(Sender: TObject);
var
  ListaProdutos: TList<TProduto>;
  ProdutoTemp: TProduto;
  i, j: Integer;
begin
  if FFornecedorAtual <> nil then
  begin
    for i := 0 to sgProdutos.RowCount - 1 do
      for j := 0 to sgProdutos.ColCount - 1 do
        sgProdutos.Cells[j, i] := '';
    with sgProdutos do
    begin
      RowCount := 2;
      Cells[0, 0] := 'Id';
      Cells[1, 0] := 'Descrição';
      Cells[2, 0] := 'Quantidade';
      Cells[3, 0] := 'Valor';
    end;
    try
      ListaProdutos := GerenciadorProdutos.CreateCriteria<TProduto>
                .Where(TLinq.Eq('Idfornecedor', FFornecedorAtual.Id)).List;
      if ListaProdutos.Count > 0 then
      begin
        sgProdutos.RowCount := ListaProdutos.Count + 1;
        for i := 0 to ListaProdutos.Count - 1 do
        begin
          ProdutoTemp := ListaProdutos[i];
          sgProdutos.Cols[0].Objects[i + 1] := ProdutoTemp;
          sgProdutos.Cells[0, i + 1] := IntToStr(ProdutoTemp.Id);
          sgProdutos.Cells[1, i + 1] := ProdutoTemp.Descricao;
          sgProdutos.Cells[2, i + 1] := FormatFloat('0.00', ProdutoTemp.Quantidade);
          sgProdutos.Cells[3, i + 1] := FormatFloat('0.00', ProdutoTemp.Valor);
        end;
      end;
    finally
      ListaProdutos.Free;
    end;
  end
  else
    ShowMessage('Selecione um fornecedor!');
end;
```

Na listagem anterior, após a verificação se existe um fornecedor selecionado (FFornecedorAtual <> nil), é declarada uma variável do tipo TList<>. A classe TList, presente na Unit Generics.Collections é utilizada para a representação de uma lista ordenada onde seus itens podem ser facilmente acessíveis através de um índice. Uma TList também permite a inserção, alteração ou remoção de qualquer item presente na lista, bem como pesquisar e classificar uma lista. Posteriormente, são utilizados dois laços para Limpar os valores presentes no StringGrid, uma vez que outra busca pode ser realizada futuramente. Na sequência, são definidos os títulos de cada coluna na primeira linha do TStringGrid com a utilização da propriedade Cells[], com o número da coluna e da linha (na forma de uma matriz). O objeto ListaProdutos recebe o resultado da consulta efetuada pelo objeto GerenciadorProdutos. Como pode ser notado, o método CreateCriteria é chamado utilizando o campo Idfornecedor e o valor do fornecedor atual selecionado, em conjunto com o método Eq do TLinq que identifica que os produtos devem ser equivalentes ao Id do Fornecedor. Em seguida, se foram encontrados produtos para o fornecedor, então o objeto ProdutoTemp é utilizado para carregar cada um dos itens da lista de acordo com a variável de controle "i", onde posteriormente as células do StringList são preenchidas de acordo com os valores retornados com a utilização da propriedade Cells[]. Para concluir, a lista é liberada da memória. A **Figura 3** apresenta o visual da aplicação após a utilização do botão btListar.

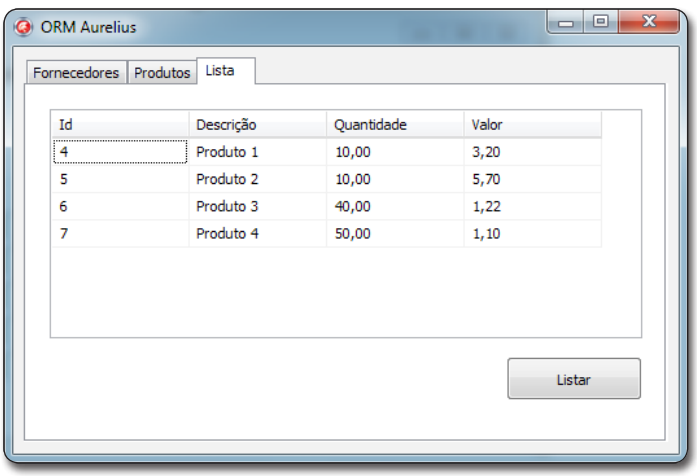


Figura 3. Visual da aplicação após a listagem dos produtos.

O Aurelius também suporta o recurso de Lazy-Loading que não foi aplicado neste projeto, mas que pode ser empregado em campos do tipo Blob, cujo objetivo é apenas carregar as informações do banco de dados se o usuário solicitar as mesmas. Um exemplo da aplicação deste recurso com o Framework é apresentado a seguir:

```
...
[Column('Imagem', [TColumnProp.Lazy])]
FImagem: TBlob;
...
```

Contudo, as funções do Framework Aurelius vão além dos recursos que foram apresentados neste projeto, uma vez que a integração com banco de dados exige diversas estruturas que devem ser empregadas sempre que houver uma necessidade mais especifica, como é o caso de uma consulta mais detalhada.

Conclusão

Os recursos da Orientação a Objetos podem ser utilizados rapidamente em projetos de aplicações baseadas em bancos de dados com o auxílio de um Framework de ORM em diversas linguagens de programação da atualidade.

O Framework Aurelius é uma Framework poderoso e completo para auxiliar o desenvolvedor a empregar ORM através do Delphi, oferecendo suporte a todas as operações necessárias para que os objetos possam ser gravados e recuperados do banco de dados de uma forma bem simples e rápida, através de seus diversos componentes. Com a utilização de atributos, as classes elaboradas pelo desenvolvedor podem ser automaticamente mapeadas, aumentando assim a produtividade no desenvolvimento da aplicação. Sendo assim, o Aurelius é uma tecnologia que vale a pena ser testada e empregada nos projetos em que o ORM é adotado, uma vez que provê os recursos necessários e é compatível com uma variedade de bancos de dados e componentes de conexão do RAD, se tornando um forte aliado no desenvolvimento de sistemas.

Giuliano Scombatti Pinto
giuliano@sygnux.com.br
É Analista Desenvolvedor da Sygnux Software (www.sygnux.com.br) localizada em Monte Alto/SP, atuando também com desenvolvimento Web e Android. Formado pela Fatec/TQ, começou a desenvolver com Delphi em 2003. Atualmente trabalha com Delphi, Java, PHP e Flash/Flex. Professor de Informática de Projetos Sociais da Prefeitura Municipal de Monte Alto/SP.

Maiores informações e download do Aurelius
http://www.tmssoftware.com/site/aurelius.asp

Maiores informações e download das ferramentas do SQLite
http://www.sqlite.org/cvstrac/wiki?p=ManagementTools

DevSpace do Autor
http://www.devmedia.com.br/space.asp?id=283274

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

RadPHP e Delphi XE 2 - Parte 2

Desenvolvendo aplicações para Android

Resumo DevMan

De que se trata o artigo:

Esta segunda parte deste minicurso tem por finalidade apresentar toda a base necessária para a elaboração de um servidor de aplicação DataSnap com o Delphi XE 2 para um ambiente heterogêneo, possibilitando assim o compartilhamento de informações entre diferentes sistemas e plataformas. Também serão apresentados os métodos e recursos necessários para que o servidor possa ser consumido por uma aplicação cliente executada no sistema operacional móvel Android, que será desenvolvida com a ferramenta RadPHP XE 2 apresentada na primeira parte deste minicurso.

Em que situação o tema é útil:

Desenvolver uma aplicação com o intuito de centralizar e tratar toda a regra de negócio da empresa ou organização, otimizando assim a manutenção e a escalabilidade do Software, permitindo ser consumida por clientes desenvolvidos em qualquer linguagem de programação e plataforma que ofereça suporte aos padrões presentes na Web que serão adotados.

Desenvolvendo aplicações para Android:

Diante do atual cenário mundial, o desenvolvimento de aplicações para ambientes móveis se tornou um grande nicho de mercado que pode acarretar em grandes benefícios financeiros aos desenvolvedores, em especial a plataforma Android por ser um ambiente "mais aberto" e acessível. Em meio a esta evolução as ferramentas presentes no RAD Studio XE 2 também evoluíram e são capazes de oferecer diversos recursos e tecnologias para serem empregadas, tanto no desenvolvimento de servidores de aplicação, através do Delphi XE 2 por exemplo, quanto no desenvolvimento de aplicações clientes para diversas plataformas, como é o caso do RadPHP XE 2, que por sua vez permite a elaboração de WebApps (Aplicações Web) capacitados a operarem através de um WebBrowser qualquer ou ainda, gerar aplicações diretamente para uma sistema operacional móvel através dos recursos da PhoneGap, uma plataforma que utiliza os padrões da Web como HTML5 e JavaScript para a criação de aplicações nativas para diversos sistemas. Nesta segunda parte deste minicurso serão abordadas as tecnologias mais importantes e necessárias para este servidor de aplicação, tais como DataSnap, arquitetura REST, notação JSON e o Framework dbExpress, apresentando também um simples banco de dados que será utilizado para este projeto, além dos métodos necessários para a manipulação das informações de forma remota, através de uma aplicação cliente desenvolvida no RadPHP para a plataforma Android ou para operar como uma aplicação Web convencional.

Este artigo faz parte de um curso

O compartilhamento de informações entre empresas, instituições e usuários atualmente é um recurso extremamente necessário para interligar, simplificar e agilizar as tarefas do cotidiano, seja ele através de uma Intranet (rede interna) ou especialmente da Internet. Em meio a este cenário, convém o desenvolvimento de Softwares que possam operar de forma automatizada e disponibilizar diversas funcionalidades aos seus usuários, independentemente de estarem presentes na empresa ou não, que sejam capacitados a oferecer uma interoperabilidade entre os mais diversos Sistemas Operacionais, permitindo assim que várias tecnologias da atualidade possam ser empregadas, tal como a plataforma móvel. Estes Softwares podem ser conhecidos como servidores de aplicação, uma vez que podem disponibilizar todas as regras de negócio da organização com o intuito de possibilitar o consumo das mesmas por parte de aplicações clientes independentes de plataforma ou ambiente operacional. Um exemplo prático desta vantagem seria o fato de um viajante ou representante de determinada organização ter acesso remoto a diversas informações

da base de dados da empresa através de um servidor de aplicação, utilizando seu dispositivo móvel como um Smartphone.

Um servidor de aplicação refere-se a uma arquitetura multicamadas, na qual o número de camadas pode variar de acordo com a necessidade e o porte do projeto em questão. A utilização de Softwares baseados em diversas camadas é muito útil, pois permite uma centralização do processamento das informações, assim como aumenta a escalabilidade, simplifica futuras manutenções, dentre outras vantagens.

Em meio a este contexto, é de obrigação do servidor de aplicação disponibilizar uma lista de serviços que podem ser acessados pelas aplicações clientes, provendo também um canal de comunicação entre ambos podendo assim oferecer recursos como autenticação e autorização de usuários. Outro ponto importante é que este servidor deve possuir uma Interface bem definida e uma estrutura adequada de objetos e componentes para prover tais funções. Interface, neste contexto, se refere aos métodos e parâmetros ou lista de recursos do servidor que são visíveis e acessíveis externamente, mas a implementação interna dos mesmos não é apresentada. De uma forma bem prática, uma interface apresenta o que uma classe pode fazer. Sendo assim, ela tem a função de disponibilizar os métodos e parâmetros necessários para que qualquer classe possa especificá-la e trabalhar de diferentes modos, aplicando alguns dos pilares da POO, como o polimorfismo por exemplo.

Quanto à execução de aplicações clientes pelos usuários, estas por sua vez são realizadas localmente em uma máquina ou equipamento, entretanto também podem ter boa parte dos processos realizados remotamente no servidor, mas sua maior porção se concentra no lado cliente, efetuando solicitações e recebendo respostas através dos métodos disponibilizados pelo servidor de aplicação com base no protocolo TCP/IP (Transmission Control Protocol/Internet Protocol que é composto por um conjunto de protocolos presentes em diversas camadas com o intuito de possibilitar a comunicação ou a transferência de informações entre equipamentos ou computadores através de uma rede de forma confiável, onde diversas questões importantes são tratadas, tal como a integridade dos dados) para possibilitar a comunicação. Este lado cliente pode ser realizado através de uma simples aplicação WebBrowser (navegador) baseada em HTML (HyperText Markup Language, uma linguagem baseada em texto e organizada em TAGS, que é interpretada por navegadores que de acordo com as instruções, efetuam a montagem visual da página) ou um Software nativo da plataforma em questão que suporte os padrões Web adotados neste contexto.

As ferramentas presentes no RAD Studio XE 2 possibilitam a criação de servidores de aplicação de maneira ágil e simplificada, aumentando assim a produtividade. Uma das tecnologias presentes no Delphi que pode ser empregada para desenvolver esse tipo de Software é o DataSnap. Este importante recurso é composto por uma diversidade de classes e componentes destinados a elaboração de aplicações que permitem a invocação de métodos remotamente, oferecendo assim a base necessária para a transferência de informações através de uma rede, especialmente a Internet. Neste cenário, outro conceito importante que deve ser aplicado para que o Software possa estar acessível por outras linguagens de programação e plataformas é o REST (Representational State Transfer) em conjunto com o JSON (JavaScript Object Notation). Em suma, o REST pode ser definido como uma arquitetura ou conceito que utiliza diversos padrões presentes na Web, como o HTTP (HyperText Transfer Protocol), tornando-o extremamente compatível com diversos ambientes. Quanto ao JSON, o mesmo pode ser definido como uma notação padrão que dentro deste contexto deve ser adotada para a transferência das informações pelo fato de ser mais leve e simples se comparado a outros padrões (Web Standards) presentes na atualidade, como é o caso do XML (Extensible Markup Language) que é uma linguagem de marcação padrão baseada em texto, estabelecida pela W3C (World Wide Web Consortium), um consórcio mundial que abrange diversas organizações, com o intuito de ser aplicada na descrição, organização e armazenamento de diversos tipos de dados, visando principalmente o ambiente da Internet. Como sugerido pelo seu próprio nome, é extensível, o que significa que o desenvolvedor pode criar e utilizar quantas TAGs julgar necessário para o seu projeto. Quanto à estrutura, um documento XML tem seu conteúdo baseado em elementos, também conhecidos como nós, onde as TAGs (etiquetas) ou delimitadores são definidos pelos caracteres "<" e ">", que por sua vez podem ou não possuir atributos, sendo os mesmos derivados de um elemento raiz e podem possuir sua própria hierarquia de elementos filhos, e assim sucessivamente.

Além das tecnologias citadas anteriormente, outro importante recurso que também pode (e deve) ser evidenciado e aplicado é o Framework dbExpress. Este último, cujo objetivo principal é permitir a conexão e manipulação de diversos bancos de dados, possui inúmeras classes e componentes que também podem ser utilizados em um servidor de aplicação baseado em DataSnap, sendo os mesmos empregados em conjunto com os métodos do servidor para o registro e obtenção das informações da base.

A tecnologia DataSnap REST

O DataSnap é uma tecnologia muito popular e importante presente no Delphi para a criação de Servidores de Aplicação, que possibilitam o compartilhamento de informações entre sistemas. Antigamente esta tecnologia era conhecida como MIDAS e foi durante muito tempo baseada em COM (Component Object Model).

Nota do DevMan

O COM (Component Object Model) foi desenvolvido com o intuito de possibilitar a utilização de objetos de forma independente da linguagem de programação, em um ambiente OO, permitindo também uma comunicação entre os processos. Sendo assim, um objeto pode ser reutilizado sem que sua implementação interna seja conhecida, uma vez que o mesmo deve possuir uma interface bem definida. Dentre este ambiente encontram-se recursos como o Active-X, documentos OLE, entre outras tecnologias.

Atualmente a tecnologia DataSnap evoluiu de forma a se tornar mais compatível com diversas plataformas. Sendo baseada no padrão de comunicações TCP/IP, recebeu várias alterações para se adequar às novas exigências do mercado. Ela também é constituída de diversos recursos nativos do próprio Delphi, o que torna possível a criação de aplicações capacitadas ao compartilhamento de suas funcionalidades através de uma rede como a Internet, possibilitando assim a invocação de métodos de forma remota e segura, constituindo uma arquitetura multicamadas, oferecendo integração com bancos de dados, entre outros recursos. A transferência das informações pode ser efetuada com base nos protocolos HTTP, HTTPS (HyperText Transfer Protocol Secure) ou TCP/IP. O HTTPS é um protocolo que efetua a implementação do SSL (Secure Socket Layer), um protocolo para implantação de segurança, sobre o HTTP. Sendo assim, os dados são criptografados para que a transferência entre o cliente e o servidor possa ser efetuada de forma segura. Entretanto, diante deste fator, a aplicação que utiliza o HTTPS tende a ser mais lenta do que uma que utilize o HTTP. Com relação a versão do Delphi XE 2, uma das novidades do DataSnap é a implementação do recurso de controle e monitoramento de conexões TCP/IP do servidor, que em outras versões exigia a elaboração de muito código fonte por parte desenvolvedor.

Outra tecnologia importante que foi mencionada e deve ser implementada no desenvolvimento de aplicações Web baseadas em DataSnap é o REST, sendo a arquitetura Cliente-Servidor adotada na World Wide Web (WWW) e uma das mais importantes da atualidade. Um servidor baseado em REST é capacitado para atender solicitações utilizando os métodos para operações de dados, GET, POST, PUT e DELETE (recuperar, atualizar, inserir e excluir), com base em um protocolo de mapeamento. Um método de um servidor REST pode ser facilmente invocado através de recursos como URI e parâmetros. O URI é baseado em uma cadeia de caracteres (string) e um protocolo, como o HTTP, com o objetivo de identificar um determinado recurso presente na Internet, ou em outras palavras, define um nome (URN – Uniform Resource Name) e/ou um endereço único (URL – Uniform Resource Locator) para que certa informação possa ser acessada e obtida. Sendo assim, uma aplicação externa pode encontrar e consumir facilmente os dados disponibilizados por outra utilizando este conceito.

Outro ponto importante desta arquitetura é que ela possibilita a utilização de qualquer tecnologia em conjunto, como HTML e XML, pelo fato de ser baseada no consumo de recursos através do protocolo HTTP, entretanto, convém sempre adotar a tecnologia que possibilita uma maior performance para a aplicação, sendo neste caso a notação JSON.

A tecnologia REST também pode ser utilizada para substituir os padrões comuns adotados em WebServices, tais como SOAP (Simple Object Access Protocol) que é um protocolo padronizado cujo objetivo é possibilitar a transferência de informações entre entidades e ambientes heterogêneos utilizando o protocolo HTTP para o transporte e o XML para a estrutura e WSDL (WebService Definition Language) que produz um documento baseado em XML que tem por objetivo descrever as funcionalidades ou métodos que estão presentes em um WebService, bem como as informações necessárias para utilizá-los, normalmente este documento pode ser facilmente obtido através do endereço do WebService com a adição dos caracteres "?wsdl". Isso, por ser mais rápida, uma vez que as mensagens em WebServices devem ser encapsuladas pelo protocolo SOAP.

Outra denominação que pode ser encontrada em aplicações baseadas nesta arquitetura, inclusive no Delphi, é o RESTful, que nada mais é que a implementação de sistemas que seguem os conceitos e restrições estabelecidos pelo REST à risca, assim como o protocolo HTTP para a transferência, maximizando a portabilidade, o desempenho, a confiabilidade, entre outros diversos fatores. Sendo assim, se a aplicação utiliza alguns conceitos desta arquitetura e viola outros, ela não pode ser considerada estritamente RESTful.

JSON e Delphi XE 2

O JSON é uma tecnologia empregada para a transferência de informações entre aplicações de uma maneira simples e com baixo consumo de recursos, sendo baseado em um subconjunto próprio de notação de objetos da linguagem de programação Client-Side JavaScript, podendo também ser aplicado para o armazenamento de informações em formato texto. Apesar de ter se originado na linguagem citada anteriormente, várias outras linguagens de programação oferecem suporte a manipulação dos recursos do JSON, tais como o Delphi e o PHP (PHP: HyperText Preprocessor). Também pode ser aplicado e indicado na substituição de documentos XML, uma vez que a interpretação dos dados de um documento com base neste recurso tende a ser mais rápida que o equivalente as mesmas informações em XML, devido a sua estrutura, permitindo também que o desenvolvedor possa efetuar a conversão de um documento para um objeto.

Quanto à estrutura, um documento JSON tem seus membros definidos pelos caracteres "{ }", constituídos por pares compostos

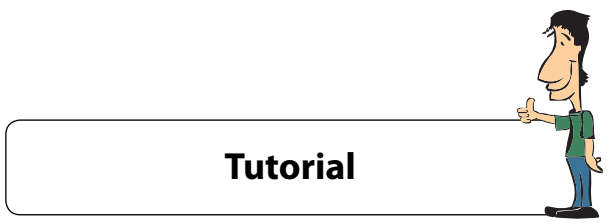
por "nome" : "valor", devendo os mesmos serem separados por "," quando houver mais de um membro. Outros caracteres importantes são os "[ ]", utilizados para a representação de um array de elementos. Um simples exemplo da estrutura de um documento JSON é apresentado a seguir:

```
{"mensagem":{"conteudo":[{"valor":"novo"},{"valor":"gravar"}]}}
```

Sendo equivalente ao documento XML demonstrado a seguir:

```
<mensagem>
  <conteudo valor="novo" />
  <conteudo valor="gravar" />
</mensagem>
```

O JSON também é o padrão adotado internamente no Delphi para o protocolo de comunicação entre um servidor baseado em DataSnap REST e uma aplicação cliente. Contudo, seus recursos podem ser facilmente utilizados para as mais diversas finalidades através das Units DBXJSON, que contém grande parte das classes utilizadas e a DBXJSONReflect, que oferece o suporte para a conversão de objetos JSON e outros tipos. A **Tabela 1** apresenta as principais classes em Delphi utilizadas para a manipulação de objetos JSON.



Banco de dados do projeto

Antes de dar início ao desenvolvimento do servidor de aplicação DataSnap, será criado um banco de dados e suas devidas tabelas para a futura elaboração e utilização dos métodos do servidor, uma vez que os recursos do dbExpress também serão empregados. Neste exemplo é desenvolvida toda a estrutura necessária para possibilitar que uma aplicação cliente que será futuramente desenvolvida em Android possa consultar uma pessoa, um item e posteriormente efetuar uma venda de maneira remota. O SGBD adotado foi o Firebird 2.1, mas outro qualquer poderia ser utilizado sem maiores problemas. Depois de criado o banco, devem ser criadas as tabelas que serão acessadas pela aplicação. A primeira tabela criada foi denominada "PESSOA", para possibilitar o armazenamento do cadastro de clientes ou fornecedores de uma empresa. A **Listagem 1** apresenta o código SQL utilizado para a criação da tabela "PESSOA".

Neste trecho a tabela PESSOA foi criada com apenas alguns campos mais necessários e uma chave primária. A próxima tabela a ser criada para este exemplo foi denominada "ITEM", com o objetivo de armazenar um produto ou serviço prestado pela empresa. A **Listagem 2** apresenta o código SQL utilizado para a tabela ITEM.

Listagem 1. Código SQL da tabela PESSOA

```
CREATE TABLE PESSOA (
  CODIGO   INTEGER NOT NULL,
  NOME     VARCHAR(50),
  ENDERECO VARCHAR(50),
  CIDADE   VARCHAR(30),
  TELEFONE VARCHAR(15)
);
ALTER TABLE PESSOA ADD CONSTRAINT PKY_PESSOA PRIMARY KEY (CODIGO);
```

Listagem 2. Código SQL da tabela ITEM

```
CREATE TABLE ITEM (
  CODIGO     INTEGER NOT NULL,
  DESCRICAO  VARCHAR(50),
  VALORCUSTO NUMERIC(9,2),
  VALORVENDA NUMERIC(9,2),
  QUANTIDADE NUMERIC(9,3),
  ATIVO      VARCHAR(1)
);
ALTER TABLE ITEM ADD CONSTRAINT PKY_ITEM PRIMARY KEY (CODIGO);
```

Concluído o processo de geração da tabela de itens, para finalizar o projeto do banco de dados, será criada a tabela de vendas, que recebeu a denominação de "VENDA" e consequentemente uma tabela de itens da venda, "VENDAITENS". A **Listagem 3** apresenta o código SQL utilizado para a tabela de vendas e a **Listagem 4** demonstra o código da tabela de itens da venda.

Classe	Descrição
TJSONObject	Esta é a classe padrão utilizada para a implementação de um objeto baseado em JSON. Dentre suas funções, merecem destaque o método ParseJSONValue, que permite a criação de um objeto com base em um byte array, e o AddPair, utilizado para a adição de um novo par ao mesmo.
TJSONArray	Esta é a classe utilizada para a implementação de um array baseado em JSON. Dentre suas funções, merecem destaque o método Add, para a adição de um objeto ao Array, e o método ToString, responsável por efetuar a serialização (ou conversão) do mesmo para uma String.
TJSONPair	Esta classe é utilizada para a implementação de um par de valores JSON, através da definição string : valor.
TJSONException	Classe empregada na manipulação de exceções que normalmente ocorrem na conversão de objetos.
TJSONNumber	Classe utilizada para a implementação de um número JSON.
TJSONTrue	Classe que permite a implementação de um valor verdadeiro para JSON.
TJSONFalse	Classe que permite a implementação de um valor falso para JSON.
TJSONNull	Classe utilizada para a implementação de um valor nulo para JSON.

Tabela 1. Classes principais do Delphi para a manipulação de objetos JSON

Criando o servidor de aplicação

O Delphi XE 2 traz consigo diversas formas práticas para a elaboração de servidores de aplicação, onde os componentes necessários e suas propriedades são automaticamente criados e definidos com a utilização dos assistentes. Para dar início a esta aplicação que será utilizada como base neste minicurso, basta criar um novo projeto DataSnap/REST através do menu File>New>Other>DataSnap Server>DataSnap REST Application. A opção "Stand-alone VCL Application" foi mantida para o desenvolvimento do servidor de aplicação com base na VCL (Visual Component Library). A **Figura 1** apresenta a janela do assistente que é exibida para a criação da aplicação DataSnap/REST.

Neste assistente é possível notar que existem algumas opções que devem ser selecionadas para a elaboração do projeto de acordo com a necessidade. A **Tabela 2** apresenta detalhadamente a função de cada uma das alternativas apresentadas.

Após avançar, a próxima etapa é a definição de um número de porta para possibilitar a futura comunicação através do protocolo TCP/IP, onde a porta 8080 é sugerida, mas nada impede que a mesma seja alterada posteriormente. Um ponto que deve ser considerado neste momento é com relação ao fato de um número de porta já estar em uso por outra aplicação, o que poderia gerar um conflito, salientando também que a

Listagem 3. Código SQL da tabela VENDA

```
CREATE TABLE VENDA (
  CODIGO     INTEGER NOT NULL,
  DATA       DATE,
  VENDEDOR   VARCHAR(30),
  TOTALVENDA NUMERIC(9,2),
  FORMAPGTO  VARCHAR(5),
  EFETIVADA  VARCHAR(1)
);

ALTER TABLE VENDA ADD CONSTRAINT PKY_VENDA PRIMARY KEY (CODIGO);
```

Listagem 4. Código SQL da tabela VENDAITENS

```
CREATE TABLE VENDAITENS (
  CODIGOVENDA  INTEGER NOT NULL,
  CODIGOITEM   INTEGER NOT NULL,
  QUANTIDADE   NUMERIC(9,3),
  VALORVENDA   NUMERIC(9,2),
  SUBTOTAL     NUMERIC(9,2)
);

ALTER TABLE VENDAITENS ADD CONSTRAINT PKY_VENDAITENS PRIMARY KEY
(CODIGOVENDA, CODIGOITEM);

ALTER TABLE VENDAITENS ADD CONSTRAINT FKY_VI_ITEM FOREIGN KEY
(CODIGOITEM) REFERENCES ITEM (CODIGO) ON UPDATE CASCADE;

ALTER TABLE VENDAITENS ADD CONSTRAINT FKY_VI_VENDA FOREIGN KEY
(CODIGOVENDA) REFERENCES VENDA (CODIGO) ON UPDATE CASCADE;
```

Nota do DevMan

O ISAPI (Internet Server Application Programming Interface) é um conjunto de recursos e APIs (Application Programming Interface) presentes no ambiente Windows para serem aplicados em servidores Web, tal como o IIS, que permite a execução de uma aplicação através do carregamento de um módulo. Para trabalhar com ISAPI, existe a necessidade de utilizar uma DLL (Dynamic Link Library), que contém os recursos que serão chamados pelo servidor.

Nota do DevMan

O IIS (Internet Information Services) é um Software do tipo servidor Web presente na plataforma Windows que pode ser utilizado para interpretar páginas ASP (Active Server Pages), módulos desenvolvidos por terceiros (como as DLLs), entre outros recursos.

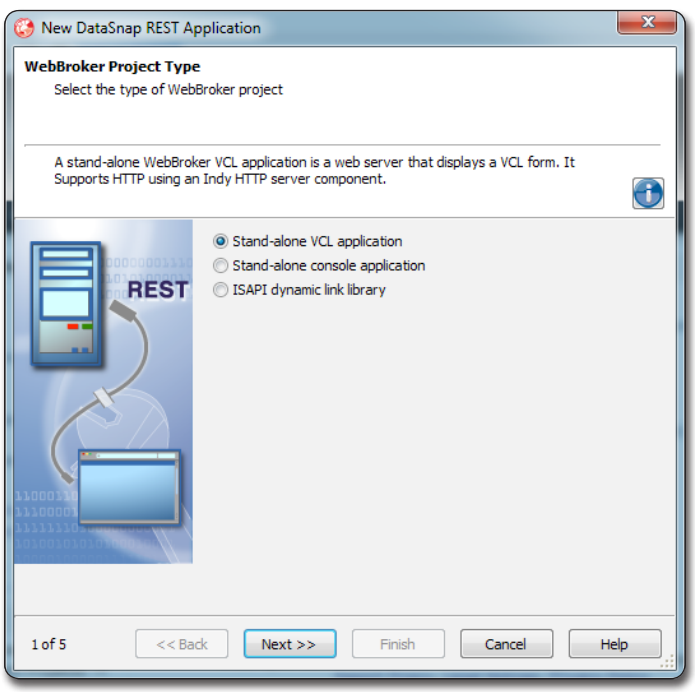


Figura 1. Assistente para a criação da Aplicação DataSnap/REST

Opção	Descrição
Stand-alone VCL Application	Utilizado para criar o servidor DataSnap utilizando os recursos da VCL, suportando o protocolo HTTP com a utilização de componentes Indy.
Stand-alone console Application	Permite a criação de um servidor com base em um console possuindo uma interface de usuário baseada em texto, também utilizando os recursos Indy para o HTTP.
ISAPI Dynamic Link Library	Permite a criação de uma biblioteca para ser utilizada pelos recursos ISAPI (Internet Server Application Programming Interface), integrados a um servidor IIS.

Tabela 2. Opções disponíveis para a geração de um servidor DataSnap

mesma deve ser liberada no Firewall, que por sua vez é a aplicação ou dispositivo responsável pelas políticas de segurança do equipamento ou da rede em geral. Na sequência foram mantidas apenas as opções já definidas (Server Methods Class), mas o desenvolvedor pode habilitar as outras se assim desejar. A opção "Mobile Connectors", utilizada para a criação dos Proxys para outras plataformas, neste exemplo é opcional mesmo que a aplicação Cliente seja destinada ao Sistema Operacional Android, pois a aplicação Cliente do Servidor de Aplicação será desenvolvida através da ferramenta RadPHP XE 2. Na tela seguinte foi selecionada a opção "TDSServerModule" e para concluir, o local de gravação do projeto. O TDSServerModule é utilizado para expor os métodos, conjunto de dados, entre outros, do servidor para a aplicação cliente. As outras opções, TDataModule e TComponent são utilizadas para, respectivamente, basear a classe do servidor em componentes não visuais e permitir a implementação da classe do servidor por parte do desenvolvedor. Após a finalização do assistente, são criados automaticamente um formulário (FormUnit1), um módulo do tipo TDSServerModule (ServerMethods1) e outro módulo do tipo TWebModule (WebModule1), bem como o conjunto de componentes necessários para a utilização do servidor de aplicação. A **Tabela 3** apresenta uma descrição sobre os objetos utilizados no servidor DataSnap presentes no WebModule.

Nota do DevMan

O TWebModule é um módulo criado automaticamente pelo Delphi quando uma aplicação Web é concebida. Ele é utilizado como um repositório para o armazenamento de diversos itens e componentes não-visuais, como TSQLDataSet e o TPageProducer. Oferece a estrutura para as respostas às requisições HTTP do servidor.

Nota do DevMan

O Proxy em um servidor DataSnap é o recurso que permite ao desenvolvedor gerar classes através do Delphi com o objetivo de serem utilizadas no desenvolvimento de aplicações em outras plataformas, como Java, Android e iOS por exemplo, permitindo assim a invocação e tratamento dos métodos remotos do servidor.

Ao concluir o assistente também é possível notar que o Delphi cria automaticamente duas funções para possibilitar o teste do servidor DataSnap, evidentemente se a opção "Sample Methods" foi mantida, sendo elas a EchoString(), que retorna o valor String com base em uma entrada, e a ReverseString(), que retorna uma String reversa com relação ao parâmetro que foi definido. Como a arquitetura REST é empregada, após iniciar o servidor este método pode ser simplesmente acessado e testado através de um navegador utilizando a seguinte URI:

```
http://localhost:8080/Datasnap/Rest/TServerMethods1/ReverseString/12345
```

Neste trecho é possível notar que é utilizado o protocolo (HTTP), em seguida o endereço do servidor (localhost, que deve ser futuramente substituído pelo endereço IP), o número da porta (8080), o endereço (REST path) definido pelo Delphi (Datasnap/Rest), o nome da classe que contém o método (TServerMethods1), o nome da função (ReverseString) e por fim, o parâmetro de entrada que neste exemplo foi definido como "12345", o que implica no retorno em JSON {"result":["54321"]}.

Com a aplicação criada o próximo passo é inserir os outros recursos necessários para o servidor, tais como componentes adicionais, métodos, etc. Basta acessar o objeto ServerMethods1 que possuirá todas as funções que serão disponibilizadas pelo servidor de aplicação e adicionar um componente TSQLConnection da aba dbExpress, que por sua vez é o responsável pela conexão com o banco de dados. Para este exemplo o arquivo dbxconnections.ini, localizado no diretório "C:\Users\Public\Documents\RAD Studio\dbExpress\9.0\" foi editado e recebeu uma nova seção que foi denominada "DADOSANDROID" para as configurações do banco. Através deste arquivo o desenvolvedor pode criar e definir os parâmetros para todos seus bancos de dados para trabalhar

Componente	Descrição
TDSServer	Este componente é o responsável pela vida útil das classes do servidor, bem como o gerenciamento do mesmo, permitindo a utilização das funções de inicialização e encerramento do serviço. Em outras palavras, é o que transforma a aplicação em um servidor. Um servidor DataSnap deve ter apenas um componente deste tipo.
TDSServerClass	Utilizado para a especificação de uma classe do servidor que contém métodos que podem ser invocados por uma aplicação cliente de forma dinâmica.
TDSHTTPWebDispatcher	Componente responsável por despachar as requisições DataSnap.
TDSProxyDispatcher	Componente utilizado para o despacho de requisições através de Proxy, destinado à plataforma móvel.
TWebFileDispatcher	Responsável pelo gerenciamento das propriedades referentes à requisição HTTP pertencente ao arquivo atual.
TDSProxyGenerator	Utilizado para a implementação de um gerador de Proxy em um servidor DataSnap.
TDSServerMetaDataProvider	Componente responsável pela implementação de um provedor de Metadados para o servidor.
TPageProducer	Componente utilizado para a conversão de modelos HTML para uma sequência de comandos também em HTML, permitindo assim a interpretação do mesmo pela aplicação Cliente ou navegador Web.

Tabela 3. Componentes do servidor DataSnap criados automaticamente

com o dbExpress de uma forma mais interessante, pois pode distribuí-lo em conjunto com a aplicação e possibilitar que algumas informações do banco de dados possam ser alteradas sem que o projeto tenha que ser recompilado. Sendo assim, para definir uma nova seção neste arquivo, o desenvolvedor pode copiar a seção já existente "FBConncetion", que contém os parâmetros destinados ao Firebird e defin-la com o nome desejado, que neste exemplo foi denominado "DADOSANDROID" e posteriormente efetuar a alteração de alguns parâmetros, conforme o trecho a seguir:

```
...
[DADOSANDROID]
DriverName=Firebird
Database=D:\ServidorDataSnapAndroid\Banco\DADOS.FDB
RoleName=RoleName
User_Name=sysdba
Password=masterkey
...
```

Neste trecho, a única alteração que foi efetuada além do nome da seção foi o parâmetro "Database", para definir o diretório onde o banco de dados está gravado. Concluído este processo, o objeto SQLConnection1 criado anteriormente deve ter sua propriedade "ConnectionName" definida como "DADOSANDROID", onde a propriedade "LoginPrompt" foi definida como False para não solicitar os dados de Login do usuário do banco de dados e a propriedade "LoadParamsOnConnect" foi alterada para True, fazendo com que os parâmetros do arquivo dbxconnections.ini sejam carregados sempre que a conexão for efetuada, lembrando que a propriedade Connected também deve ser definida como True para possibilitar a conexão.

Terminada a configuração da conexão com o banco, o próximo objetivo é iniciar a elaboração dos métodos do servidor que serão acessados pelas aplicações clientes, bem como aqueles que não serão acessíveis. Como os métodos efetuarão o acesso ao banco, alguns utilizarão os recursos da classe TSQLQuery, presente na Unit Data.SqlExpr e que faz parte do Framework dbExpress. Esta classe provê as funções necessárias para efetuar uma consulta (Query) a um banco de dados, armazenando a informação retornada, ou ser empregada para executar ações ao mesmo através de comandos SQL (INSERT, UPDATE, DELETE, ALTER TABLE, etc), devendo a mesma ser interligada a um objeto TSQLConnection. Entretanto, um objeto do tipo TSQLQuery é unidirecional e sua navegação pode ser efetuada através dos comandos First (Primeiro) e Next (Próximo). Outro ponto que deve ser evidenciado é que este objeto não possui autonomia para efetuar edições (Edit) diretamente em registros obtidos de uma consulta, mas isso se torna possível com a utilização de componentes adicionais como um provedor de dados e um ClientDataSet, se assim for necessário.

A primeira função criada baseada na utilização de um objeto TSQLQuery tem o simples objetivo de gerar um novo código toda vez que for chamada para qualquer tabela e campo, conforme seus parâmetros. Neste momento, o ideal é definir quais os métodos privados e públicos desta classe, efetuando assim o encapsulamento das informações e apresentado apenas uma interface com os comandos públicos. Esta função foi definida na seção private, uma vez que a mesma só será acessível e necessária para a classe atual e não deve ser disponibilizada como um método acessível remotamente pelo servidor. A **Listagem 5** apresenta a função GetNovoCodigo().

Listagem 5. Função GetNovoCodigo()

```
function TServerMethods1.GetNovoCodigo(Tabela, Campo: string): Integer;
var
  Qry: TSQLQuery;
begin
  Qry := TSQLQuery.Create(nil);
  try
    with Qry do
    begin
      SQLConnection := SQLConnection1;
      SQL.Text := ' SELECT MAX('+ Campo +') '+
            ' FROM '+ Tabela;
      Open;
      Result := Fields[0].AsInteger + 1;
      Close;
    end;
  finally
    FreeAndNil(Qry);
  end;
end;
```

Neste trecho de código pode-se notar que a função recebe dois parâmetros, sendo eles a tabela e o campo da mesma para que um novo código sequencial possa ser elaborado. Sendo assim, a classe TSQLQuery é instanciada para a criação de um objeto do tipo Query, pertencente ao Framework dbExpress. Posteriormente, a propriedade SQLConnection do mesmo é definida com o objeto de conexão criado anteriormente, o SQLConnection1. Como pode ser observado, no código SQL é efetuado um "SELECT MAX()" com o intuito de obter o maior número presente na tabela para a geração de um novo código adicionando-se 1 para a chave primária, que por sua vez é retornado pela função (Result).

Nota

Para fins didáticos utilizamos o SELECT MAX para geração de chaves primárias, contudo o ideal poderia ser utilizar o que o banco de dados em uso oferece, como por exemplo Generators/Sequences, tipos de dado auto-incremento ou até mesmo o uso de tabelas auxiliares

A próxima etapa neste projeto é criar a função que permitirá o cadastramento de uma Pessoa através da aplicação Cliente. Ela foi denominada InserirPessoa() e teve alguns parâmetros definidos para que possa receber as informações necessárias para o cadastramento dos dados, bem como retornará uma mensagem e um valor booleano de acordo com a execução da mesma. A **Listagem 6** apresenta o código da função InserirPessoa().

Neste trecho de código, são criados quatro parâmetros de entrada (Nome, Ender, Cid e Tel) e um de saída (mensagem) através do comando out, além do retorno booleano informando se a mesma foi efetuada com sucesso ou não. Como pôde ser

observado, caso não ocorram erros uma mensagem de processo efetuado é gravada no parâmetro mensagem e a função retorna um valor verdadeiro, caso contrário uma mensagem de exceção é elaborada e o retorno enviado pela função tem seu valor definido como falso.

Listagem 6. Função InserirPessoa()

```
function TServerMethods1.InserirPessoa(Nome, Ender, Cid, Tel: string;
 out Mensagem: string): Boolean;
var
  Qry: TSQLQuery;
begin
  Qry := TSQLQuery.Create(nil);
  try
    with Qry do
    begin
      SQLConnection := SQLConnection1;
      SQL.Text := ' INSERT INTO PESSOA '+
                  ' VALUES ('+ IntToStr(GetNovoCodigo('PESSOA','CODIGO')) +','+
                  QuotedStr(Nome) +','+ QuotedStr(Ender) +','+
                  QuotedStr(Cid) +','+ QuotedStr(Tel) +');';
      ExecSQL;
      Close;
    end;
    Mensagem := 'Processo Efetuado!';
    Result := True;
  except
    on e: Exception do
    begin
      Mensagem := 'Houve um erro: ' + e.Message;
      Result := False;
    end;
  end;
  FreeAndNil(Qry);
end;
```

O próximo método elaborado tem o objetivo de obter todas as pessoas cadastradas na base de dados ou retornar as que possuem seu nome de acordo com o parâmetro estabelecido através do comando SQL SELECT em conjunto com LIKE. Nesta função, o retorno será um objeto do tipo TDBXReader presente no Framework dbExpress. Para tanto a Unit Data.DBXCommon deve ser adicionada na seção uses para que as classes sejam acessíveis. Ao criar uma instância da classe TDBXReader, o desenvolvedor conta com um leitor unidirecional para os registros de um banco de dados deste Framework, sendo assim, a navegação pelas informações desde a primeira linha (First) deve ser efetuada através do comando Next e assim sucessivamente. Para o acesso aos valores a propriedade Value deve ser utilizada e pelo fato de ser um Array, os dados podem ser obtidos informando o nome da coluna ou o índice referente à posição da mesma. Outra classe que também é utilizada e muito importante para a execução deste exemplo (pois retorna seu valor em TDBXReader) é a TDBXCommand. Esta classe possibilita a execução de instruções SQL ou procedimentos armazenados. Um objeto TDBXCommand pode executar diversos tipos de comandos de acordo com a classe TDBXCommandTypes, onde a definição do mesmo é efetuada através da propriedade CommandType. Também é possível criar parâmetros para este objeto utilizando o método TDBXCommand.CreateParameter(). Após definido o comando que será utilizado o desenvolvedor pode atribuir a tarefa que será efetuada através da propriedade Text do mesmo e em seguida, chamar o método Prepare para preparar a execução do comando que, por sua vez, é realizada pelo método ExecuteQuery(). A **Listagem 7** apresenta a função GetPessoas() utilizando os recursos mencionados anteriormente.

Listagem 7. Função GetPessoas()

```
function TServerMethods1.GetPessoas(Nome: string): TDBXReader;
var
  Comando: TDBXCommand;
  Filtro: string;
begin
  if Trim(Nome) <> '' then
    Filtro := ' WHERE NOME LIKE '+ QuotedStr('%'+ Nome + '%');
    Comando := SQLConnection1.DBXConnection.CreateCommand;
    Comando.CommandType := TDBXCommandTypes.DbxSQL;
    Comando.Text := ' SELECT CODIGO, NOME '+
                    ' FROM PESSOA '+ Filtro +' ORDER BY CODIGO ';
  if not Comando.IsPrepared then
    Comando.Prepare;
  Result := Comando.ExecuteQuery;
end;
```

Nesta função, se o parâmetro for definido com algum valor (Trim(Nome) <> ''), então é criado um filtro baseado em LIKE para o comando SQL SELECT, caso contrário todas as pessoas cadastradas são retornadas pela mesma. Posteriormente o comando é criado para a conexão SQLConnecton1 através do método DBXConnection.CreateCommand(). Em seguida é definido o tipo do comando para SQL através de TDBXCommandTypes.DbxSQL e a propriedade Text recebe as instruções necessárias para efetuar a consulta. Para concluir, o comando é executado e seu retorno é utilizado como resultado da função (Result).

O próximo passo é criar as funções para os Itens, que serão semelhantes a estes, portanto este processo fica como uma tarefa extra para a familiarização com os comandos.

Recursos adicionais opcionais

No desenvolvimento de um servidor de aplicação DataSnap REST com o Delphi XE 2 também existem outros recursos que não foram aplicados neste exemplo mas que podem ser utilizados facilmente, tais como a autenticação e autorização de usuários no momento em que os mesmos tentam consumir os recursos do servidor. Durante a criação do servidor, através do assistente, o desenvolvedor deve ativar as opções "Authentication" e "Autorization". O componente responsável pelo gerenciamento deste recurso é o TDSAuthenticationManager presente na Unit DSAuth, que é criado automaticamente pelo assistente caso as alternativas descritas anteriormente estiverem marcadas. Ele possui dois eventos, o OnUserAuthenticate e o OnUserAutorize. O evento OnUserAuthenticate é responsável pelo gerenciamento de autenticações, sendo chamado no momento em que os usuários tentam se conectar ao servidor,

salientando que em aplicações REST esta invocação ocorre a cada nova seção. Quanto ao evento OnUserAutorize, o mesmo é empregado para gerenciar a autorização que determinado usuário tem para acessar o método invocado. Entretanto, ao trabalhar com estes recursos, o desenvolvedor deve se lembrar de empregar toda estrutura necessária na aplicação cliente para a autenticação dos usuários, uma vez que os recursos estarão bloqueados no servidor de aplicação. Contudo, se não desejar utilizar tal forma apresentada para a autenticação o desenvolvedor ainda pode implementar manualmente seus próprios códigos com este objetivo, tratando a verificação de usuários conforme achar mais adequado para o projeto.

Outro recurso adicional que não foi aplicado mas também é muito útil e pode ser adicionado durante o uso do assistente, é o componente TDSProxyGenerator presente na Unit Datasnap. DSClientMetadata, através da opção Mobile Connectors. Ele é o responsável pela geração dos Proxys para o servidor DataSnap, lembrando que Proxys neste contexto são os recursos ou classes que o Delphi gera automaticamente para que o desenvolvedor possa criar facilmente aplicações clientes em outras linguagens de programação, mas não são uma obrigatoriedade.

Criando uma WebApp no RadPHP para testar o projeto

Conforme será abordado futuramente, este servidor de aplicação será consumido por um software cliente para plataforma móvel Android, entretanto o desenvolvedor também pode facilmente consumi-lo através de outros ambientes, utilizando navegadores, por exemplo. Como foi apresentado no primeiro artigo deste minicurso, uma ferramenta extremamente útil e de alta produção para a criação de WebApps é o RadPHP, que é parte do RAD Studio XE 2 e utiliza as linguagens de programação PHP, HTML e JavaScript como base para o desenvolvimento.

Para dar início a esta etapa do projeto, após abrir o RadPHP, deve-se criar uma nova aplicação RPCL (RadPHP Component Library) através do menu Arquivo>Novo>Aplicação RPCL – PHP.

Nota do DevMan

RPCL é uma biblioteca de classes que encapsula os scripts e bibliotecas PHP open source, incluindo qooxdoo, ADOdb, DynAPI, Smarty, xajax e JSCalendar e outras para criar uma biblioteca de componentes visuais que suporta um desenvolvimento no modelo arrastar e soltar. Inspirada pela VCL do Delphi, sua arquitetura é 100% escrita em PHP. Desenvolvedores podem criar e integrar componentes no IDE e estender o que já existe para sua necessidade. A biblioteca de componentes também inclui suporte para o framework Zend, Facebook e Google Maps.

Neste exemplo, o método criado anteriormente GetPessoas() será consumido pela WebApp, portanto foram inseridos um botão (SpeedButton) para chamar o método e uma lista (ListBox), que exibirá os valores retornados. O próximo passo é implementar os recursos necessários para consumir os métodos disponibilizados pelo servidor de aplicação. O RadPHP possui um assistente que permite a importação de tais métodos de uma forma simples e diretamente do servidor DataSnap. Para acessá-lo, basta ir até o menu Arquivo>Novo>Outro>Projetos RadPHP>Arquivos PHP>Módulo cliente DataSnap REST. Entretanto, este recurso é opcional pois se desejar, o desenvolvedor pode implementar o consumo do servidor também de forma manual através do RadPHP. A **Figura 2** apresenta a janela do assistente de importação utilizado para gerar automaticamente as classes necessárias para o consumo do servidor.

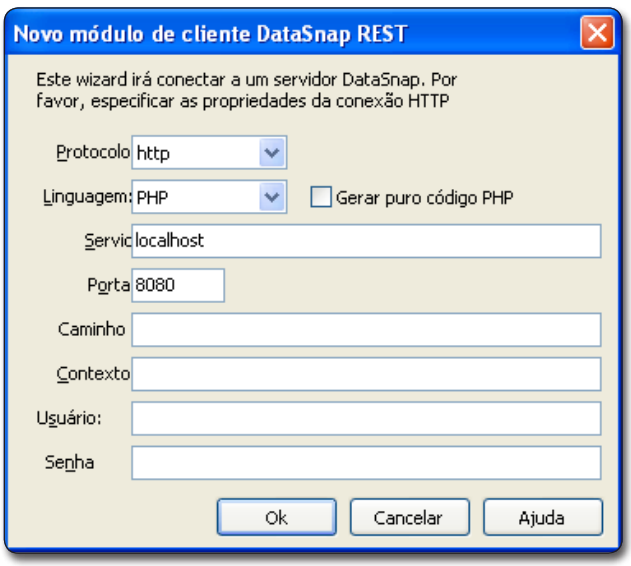


Figura 2. Assistente para importação de recursos do servidor DataSnap/REST

Este assistente conta com alguns campos que devem ser preenchidos para que a importação seja efetuada, tais como o protocolo, que pode ser HTTP ou HTTPS, a linguagem, PHP ou JavaScript, o Servidor, que neste exemplo foi atribuído localhost (o endereço da máquina local, equivalente à 127.0.0.1), e o número da porta, 8080, a mesma definida anteriormente, entretanto, tais valores podem ser alterados futuramente. Com a conclusão do assistente, serão criados dois novos objetos, o ClientClassUnit1, que contém os métodos importados, e o módulo ClientModuleUnit1, que contém o objeto DSRestConnection, que por sua vez é o responsável pela conexão com o servidor DataSnap e possui as informações de integração definidas anteriormente, tais como o host e a porta. Sendo assim, na Unit principal do projeto, que neste exemplo foi denominada "UnitPrincipal.php", deve-se adicionar uma referência para o ClientModuleUnit1A **Listagem 8** apresenta o código fonte que foi atribuído à classe FormPrincipal e ao evento OnClick do SpeedButton1 desta Unit para concluir o desenvolvimento da aplicação cliente WebApp.

Neste trecho de código é possível observar que após a declaração da classe FormPrincipal1, são geradas automaticamente as referências aos objetos da aplicação, $SpeedButton1 e $ListBox1. Posteriormente no evento OnClick() do SpeedButton1, é efetuada a declaração global do objeto ClientDataModule1, permitindo assim o acesso ao mesmo. Em seguida o método GetPessoas() é chamado através do módulo e o resultado de sua execução é

armazenado no objeto $obj. Um ponto que deve ser ressaltado neste momento é a utilização da função var_dump() em tempo de desenvolvimento (conforme o código comentado), cujo objetivo é apresentar o conteúdo de um objeto ou variável, sendo muito útil neste contexto por possibilitar ao desenvolvedor visualizar a resposta retornada pelo servidor DataSnap e assim tratá-la da melhor forma ou da maneira desejada. Logo em seguida é criada uma variável ($contador) que recebe o número de pessoas retornadas, através do uso da função count() em conjunto com o objeto, contanto o número de códigos ($obj->{'CODIGO'}). Posteriormente é efetuado um laço for() com base no valor da variável $contador e para concluir, o objeto ListBox ($ListBox1) adiciona cada valor presente no array ($obj->{'CODIGO'}[$i]) de acordo com a variável de controle $i.

Listagem 8. Código da classe FormPrincipal do WebApp

```
class FormPrincipal extends Page
{
  public $SpeedButton1 = null;
  public $ListBox1 = null;
  function SpeedButton1Click($sender, $params)
  {
    global $ClientModuleDataModule1;
    $obj = $ClientModuleDataModule1->ServerMethods1Client->
    GetPessoas('')->result;
    $this->ListBox1->Clear();
    //var_dump($obj);
    $contador = count($obj->{'CODIGO'});
    for ($i = 0; $i < $contador; $i ++) {
      $this->ListBox1->AddItem($obj->{'CODIGO'}[$i] .' - '. $obj->{'NOME'}[$i]);
    }
  }
}
```

Concluído este exemplo é possível notar que o desenvolvimento de aplicações clientes do tipo WebApp para um servidor DataSnap é uma tarefa fácil e simples, entretanto para a geração de uma aplicação para a plataforma Android outros fatores deverão ser levados em consideração, sendo os mesmos abordados no próximo artigo deste minicurso. WebApp (Web Application) é o termo empregado para definir uma aplicação que é projetada e capacitada para a execução em um navegador, tanto para uma rede interna quanto para a Internet, que se comunica e tem seu processamento efetuado em um servidor, o que implica que qualquer ambiente operacional que disponha de um WebBrowser e recursos de rede poderá ser compatível com a aplicação, evidentemente levando em consideração as tecnologias empregadas na WebApp, uma vez que navegadores obsoletos podem não oferecer total suporte aos novos recursos disponíveis.

Conclusão

Atualmente é fácil notar que o mercado móvel cresce a cada dia. As pessoas e organizações utilizam cada vez mais os recursos oferecidos pelas plataformas Android, iOS e etc para possibilitar a comunicação instantânea e o acesso as suas informações e aplicações favoritas. Diante deste fator, este mercado se torna extremamente potencial para o investimento por parte de desenvolvedores e Software Houses com o intuito de oferecer aplicações cada vez mais robustas e que apresentam um grande diferencial com relação a seus concorrentes, uma vez que as mesmas podem possibilitar o acesso a uma diversidade de recursos utilizando uma rede como a Internet.

O conceito de servidor de aplicação é muito importante neste contexto para expandir as possibilidades de uma organização e permitir a interoperabilidade entre aplicações clientes.

Nesta segunda parte deste minicurso foram abordadas diversas tecnologias que, direta ou indiretamente, são aplicadas para possibilitar a criação de um simples Servidor de Aplicação baseado em DataSnap, capaz de prover recursos para a plataforma foco deste projeto, que é o Sistema Operacional móvel Android, mas que são a base para a criação de aplicações complexas para qualquer plataforma ou linguagem de programação que for desejada.

Giuliano Scombatti Pinto
giuliano@sygnux.com.br
É Analista Desenvolvedor da Sygnux Software (www.sygnux.com.br) localizada em Monte Alto/SP, atuando também com desenvolvimento Web e Android. Formado pela Fatec/TQ, começou a desenvolver com Delphi em 2003. Atualmente trabalha com Delphi, Java, PHP e Flash/Flex. Professor de Informática de Projetos Sociais da Prefeitura Municipal de Monte Alto/SP.

Site oficial e manual do PHP
http://www.php.net/manual/pt_BR/

Maiores informações sobre JSON
http://www.json.org

DevSpace do Autor
http://www.devmedia.com.br/space.asp?id=283274

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

dbExpress de A a Z - Parte 1

Uma visão inicial de seu funcionamento

Este artigo faz parte de um curso

Quando a palavra é conectividade o Delphi consegue nos oferecer uma infinidade de oportunidades para trabalharmos com as mais diversas formas presentes na ferramenta. Podemos conectar com bancos de dados através de drivers padrões, utilizar drivers de terceiros, componentes de terceiros, conexões com Cliente/Servidor, mas sempre a questão é: "qual é a melhor forma de conectar, a mais fácil ou a mais robusta, até mesmo aquela que irei ter o melhor aproveitamento de código?".

O dbExpress vem de encontro a essas questões. Ele tem a finalidade fornecer acesso a dados dentro do Delphi. Utilizando a estrutura de drivers nativos ou aplicativos gerenciados, possuindo também componentes para a VCL.

Uma das facilidades de trabalhar com o dbExpress é que além de ser totalmente integrado ao Delphi, os drivers que estão disponíveis atendem uma grande leva de bancos de dados comerciais disponíveis atualmente no mercado, porém, se necessário for, é possível integrar outros drivers para que o Framework possa visualizar outros bancos de dados.

O dbExpress também é útil em relação a grandes proporções de dados e acessos simultâneos permitindo fácil escalabilidade e algoritmos de segurança, tais como Failover e Load Balance. Tudo isso integrado a um único Framework cuja forma de trabalho segue uma mesma linha. Aprendendo um único conceito é possível desenvolver do mais simples ao mais robusto servidor de dados e ponto de acesso a dados. Para melhorar o aprendizado vamos explorar as funcionalidades do dbExpress em artigos distintos, abordando todas

Resumo DevMan

De que se trata o artigo:

Instruir o leitor a trabalhar com as melhores práticas em relação ao acesso a dados dentro do Delphi, utilizando o Framework dbExpress 4.0 e trabalhando com conceitos propostos pela Embarcadero Brasil. Veremos desde como acessar os dados, efetuar o CRUD, pesquisas, filtros, personalização de visualização, arquitetura multicamadas, Logs, Tratamento de erros, execuções de grande porte e manipulação de estrutura de tabelas (DDL).

Em que situação o tema é útil:

Manipular dados de forma simples, rápida e prática onde em qualquer tipo de aplicativo comercial podemos aproveitar os conceitos aqui abordados. A partir deste, poderemos criar aplicativos para integração com bancos de dados como: Oracle, SQL Server, MySQL, Firebird e diversos outros. Além de podermos disponibilizar nossos dados e processamentos através de um servidor de aplicação e conectar nossos recursos através do Cloud Computing, conectar com dispositivos móveis (Android, IPhone, Windows Phone) e termos uma escalabilidade maior na comunicação de informações.

dbExpress de A a Z - Parte 1:

Nesta série de artigos iremos abordar a suíte de conectividade dbExpress que ao longo dos anos sofreu grandes transformações, tornando-se hoje um Framework indispensável para quem deseja criar aplicações que possuam multiconectividade e performance sem grandes esforços. Além das vantagens abordadas, o dbExpress possui uma integração simples e ágil com a tecnologia DataSnap, que permite a criação de aplicações multicamadas, o que hoje permite a integração entre aplicações heterogêneas, como WebSites e dispositivos móveis. Abordaremos os princípios do dbExpress, comentando seu funcionamento de uma maneira completa e abrangente.

as possibilidades que esta ferramenta nos oferece. Por ser uma recomendação dentro dos treinamentos oficiais da Embarcadero Brasil, vamos seguir alguns passos que são propostos durante os treinamentos. Passando pelo simples CRUD até chegarmos à estrutura multicamadas.

Neste primeiro artigo o conceito geral do dbExpress será abordado e chegaremos ao término deste com um CRUD sendo implementado em um banco SQL Server.

Nota do DevMan

Failover é a habilidade de uma máquina, equipamento ou Software de transferir informações de um componente que está em falha para um alternativo, sem interromper o fluxo do processo. Por exemplo, digamos que temos um servidor DataSnap em funcionamento, contudo, a máquina para de responder. Automaticamente as conexões deveriam ser redirecionadas para outra máquina, seja de Backup ou alternativa, sem que o usuário final perceba.

Nota do DevMan

Load Balance é o processo de analisar a carga de uma determina máquina recebe e distribuir essa carga em outras que estejam disponíveis. Esta é uma técnica muito comum em servidores Web, que devem controlar a quantidade de requisições que são feitas para que o tempo de resposta não seja longo.

Desvendando o trabalho com o dbExpress

O dbExpress nada mais é que um conjunto de classes que estão contidas na unit (DbxCommon). Classes estas que podem ser utilizadas para acesso direto aos dados através dos drivers padrões para bancos como SQL Server, Oracle, MySQL, Firebird e diversos outros. Ou até mesmo para acesso via componentes da VCL.

Um grande benefício ao utilizar este Framework é que o trabalho com ele é extremamente simples, além de conseguir sanar todo e qualquer tipo de execução que temos quando falamos de bancos de dados, conseguimos efetuar comandos SQL do mais simples aos mais complexos e robustos, tudo com uma agilidade sem comparativos. Iremos ver o dbExpress trabalhando em conjunto com a VCL, utilizando os componentes padrões que possuímos para conectar e manipular os nossos dados, cada um deles será abordado e terá seu uso exemplificado e suas propriedades mais importantes explicadas.

Veremos aqui que o conceito de acesso a dados simples é base para o qualquer outro tipo de acesso, logo necessitaremos conhecer poucos componentes da VCL assim como poucas classes do Framework para podermos fazer uma infinidade de processos.

Os componentes básicos do dbExpress na VCL

Quando falamos em acesso a algum tipo de banco de dados, estamos falando de uma forma bastante genérica onde qualquer item que armazene diversas informações pode se tornar um fornecedor de informações em potencial.

O dbExpress tem a capacidade de tornar desde um SGBD a arquivos tipo texto e XML uma fonte válida de dados, onde pode-se efetuar inserções, alterações, exclusões, pesquisas, filtros e o que mais houver de necessidade dentro do Delphi, salvo às limitações do próprio arquivo ou SGBD que estiver sendo a fonte destas informações.

O primeiro componente que irá compor o nosso exemplo será o TSQLConnection (**Figura 1**), este é um dos mais importantes componentes que temos dentro do Framework, pois é através dele que tudo acontece. Ele é a nossa conexão com o banco de dados. Está situado na paleta dbExpress do Delphi e possui as seguintes finalidades:

1. Ser a ponte de conexão entre o Delphi e o SGBD escolhido ou Servidor de aplicativos DataSnap;
2. Transportar as informações e executá-las no SGBD.

Figura 1. TSQLConnection

Para trabalhar com este componente precisamos conhecer algumas propriedades e alguns métodos importantes. O SQLConnection possui algumas propriedades que são encadeadas, começando no **ConnectionName** e terminando no **VendorLib**. Estas propriedades que podem ser observadas no Object Inspector fazem referência ao tipo de conexão que será estabelecida e quais são os drivers que serão utilizados. Como dito anteriormente, o dbExpress possui os próprios drivers de acesso a dados, porém também tem a capacidade de trabalhar com drivers produzidos por terceiros. Para que isto seja possível basta apenas efetuar a inclusão do mesmo nas respectivas propriedades dispostas no Object Inspector.

Dentre os diversos métodos públicos que possuímos no SQLConnection, temos o ExecuteDirect que tem a capacidade de executar comandos diretos dentro do banco, sem a necessidade de nenhum outro item para processamento, basta informamos o comando que o SQLConnection se encarregará de executar o mesmo. Além deste possuímos outros tantos métodos que estão disponíveis que vão desde iniciar uma transação a registrar métodos de monitoramento de execuções, os chamados (Traces).

Utilidades e curiosidades sobre o TSQLConnection

Este é definitivamente um tipo de "canivete suíço" que possuímos no Delphi e vale fazer a ressalva de alguns itens que ele possui. Um dos mais simples que é pouco utilizado é a capacidade de efetuar Pool de conexão tudo através da propriedade **DelegateConnection**. Um Pool de conexão é um repositório que guarda em uma lista as conexões que estão abertas e que podem ser reutilizadas. Isso é justificável porque criar um objeto de conexão é uma operação custosa. Diz-se custosa, pois envolve algumas operações que muitas vezes não damos atenção, mas acontecem. Por exemplo, exige uma conexão remota com um servidor que exige um teste de aceitação entre as máquinas envolvidas, a String de conexão passa por uma validação e se estiver correta então é autenticada no servidor. Assim, se toda vez que um sistema precisasse acessar um banco de dados fosse necessário criar uma conexão, as operações se tornariam lentas. A finalidade do Pool de conexões é justamente essa. Ao iniciar uma conexão ela pode ser inserida nesse Pool e mantida nele por um tempo, quando uma segunda requisição de acesso é identificada, é verificada se ela confere (String de conexão) com alguma já existente nesse Pool e caso exista combinação, a mesma é retornada, assim, economiza-se tempo e recursos.

Outro item curioso que possuímos neste componente é que internamente ele possui um "segundo" componente de conexão do tipo TDBXConnection que é utilizado para as conexões com servidores DataSnap, além de ser ele o responsável por gerar classes de Proxy e executar comandos mais aprimorados que podem chegar até a trabalhar diretamente com o escopo de criação de tabelas (Comandos DDL) no banco, onde soluções Delphi podem criar tabelas e relacionamentos dentro de qualquer tipo de banco sem que o desenvolvedor se preocupe em digitar uma linha se quer de comando SQL específico.

Armazenamento de comandos SQL

Agora que já possuímos a conexão com o banco e já sabemos o conceito geral para o que podemos utilizar esta conexão, iremos tratar agora dos comandos SQL.

Ao utilizarmos o Framework dbExpress possuímos uma facilidade onde a própria ferramenta se encarrega em entender e resolver as nossas manipulações de dados. Informamos o comando de seleção básico apenas e o restante dos comandos: Inserção, Atualização e Exclusão o DBX gera e manipula manualmente.

Assim o processamento do CRUD fica mais fácil durante todo seu uso. O componente dentro da VCL que nos auxilia neste processo é descendente de TDataSet, que neste caso será o TSQLDataSet (**Figura 2**).

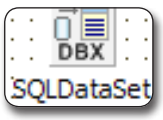


Figura 2. TSQLDataSet

Vale dizer que dentro do Delphi possuímos alguns outros que também descendem do TDataSet e que podem ser facilmente utilizados para a integração com o DBX. O TSQLDataSet foi escolhido pelo fato de ser um componente mais fácil de ser configurado e não necessitar especificamente de que o desenvolvedor digite o comando SQL, todo o texto que irá compor o comando pode ser gerado através do assistente que existe na propriedade **CommandText**.

O TSQLDataSet é basicamente um conjunto de dados unidirecional resultante de um comando SQL executado no banco, comando este que pode ser uma consulta SELECT ou o resultado de um procedimento do banco. Seu comportamento é unidirecional porque só é possível navegar para frente nesse conjunto de dados. Não é possível avançar um registro e posteriormente retornar. Para que este componente consiga funcionar é necessário estar vinculado a algum servidor de dados através da propriedade SQLConnection, o que nos leva a entender o começo da "linha de execução" do dbExpress.

Utilidades e curiosidades sobre o TSQLDataSet

Este componente nada mais é do que um "assistente" que irá conter nossos comandos que podem ser parametrizados, onde estes parâmetros podem ser passados em tempo de execução, tornando as sentenças SQL mais dinâmicas e mais fáceis de serem reaproveitadas. Os parâmetros de SQLDataSet são precedidos do ":" (dois pontos) e podem ser configurados através da propriedade Params que é uma lista de parâmetros. O SQLDataSet quando requisitado para executar um comando no banco, irá preparar o SQL modificando os parâmetros pelo valor que foi informado em tempo de execução. Então passará o comando SQL já preparado para o SQLConnection executar no servidor de dados através do comando **ExecuteDirect** ou **ExecSQL**, dependendo da situação. É devida a essa capacidade que, para efetuarmos qualquer tipo de manipulação de dados dentro do Delphi basta fornecermos o comando de seleção que todo o restante é preparado dinamicamente e executado de forma simples e transparente.

Gerando os SQLs

O armazenamento das informações é feito no SQLDataSet, que por sua vez passa os comandos para o SQLConnection executar, mas como dito anteriormente estes são apenas uma parte do que precisamos para gerar nossos comandos de Seleção, Inserção, Atualização e Exclusão.

É para isso que possuímos o DataSetProvider (**Figura 3**), este é responsável por resolver as atualizações em memória e gerar os comandos que serão executados no banco.

Figura 3. DataSetProvider

É por ele que definimos como iremos enviar os dados para o banco. Ele se conecta a um TDataSet através da sua propriedade DataSet e com isso consegue gerar atualizações para uma ou mais tabelas a ele vinculadas, suportando atualizações simples até casos de mestre-detalhe que é a situação onde temos um relacionamento de dependência entre DataSets, algo como Pedido e Itens Pedidos, ao salvar um Pedido seus Itens devem ser salvos automaticamente, todos podendo ser transitados de forma ágil e sem a necessidade da interferência externa.

O DataSetProvider é um componente capaz de gerir não só os comandos de CRUD, mas é através dele que também conseguimos gerar funções simples como LOG, onde em apenas um evento já podemos registrar tudo que se passa no sistema. Porém as funcionalidades e as vantagens do DataSetProvider vão além de gerar os SQL através de si, também podemos configurar como uma tabela se portará quando representada em memória. Por exemplo, se possuirmos uma tabela que seja necessária apenas para pesquisa, podemos deixar desabilitadas as opções de edição, exclusão ou inserção. É neste componente que faremos este tipo de configuração, através das propriedades vinculadas em Options.

Utilidades e curiosidades sobre o DataSetProvider

O DataSetprovider também é o provedor de informações quando estamos falando da arquitetura DataSnap, é através dele que podemos enviar nossas tabelas de um servidor para um cliente, criando assim uma arquitetura capaz de suportar diversos dados sendo transmitidos e compartilhados com vários clientes ao mesmo tempo, mais adiante iremos demonstrar como utilizar o DataSetProvider para também servir informações que poderão ser transmitidas através de protocolos TCP/IP, HTTP e HTTPS.

Outra funcionalidade pouco conhecida do DataSetProvider é a capacidade do mesmo de servir de ponto seguro ("Save Point") dentro do Delphi, o que quer dizer que é através deste que conseguimos manipular todas as informações dos DataSets que estão populados, tendo acesso às informações que vieram do banco e as que o usuário manipulou. Com isso passamos a saber quais dados foram modificados, qual a quantidade de modificações que foram feitas e até mesmo podemos criar regras de segurança em nossos aplicativos para que o usuário nunca saia de uma tela sem antes salvar as informações que foram modificadas etc.

Os dados em memória

Os dados que são conectados pelo SQLConnection, selecionados pelo SQLDataSet e monitorados pelo DataSetProvider são trazidos para a memória através do ClientDataSet (**Figura 4**).

Figura 4. ClientDataSet

Este componente é a representação da tabela que está disponível no banco e no Delphi para que o usuário possa manipular. Este se vincula ao DataSetProvider de várias formas, porém a básica e inicial que veremos é através da propriedade **ProviderName** onde temos por aqui uma referência simples de tabela no banco e tabela em memória. O ClientDataSet é um componente extremamente robusto, podendo trabalhar com informações e definições que vem de uma tabela no banco, ou até mesmo ser ele seu próprio "provedor" de informações onde o desenvolvedor pode definir uma estrutura de tabela virtual para que possa salvar isso posteriormente em arquivos (TXT e XML).

O ClientDataSet possui os seus "Fields" que são a representação dos campos da tabela, é com estes que podemos configurar quais serão utilizados pelo usuário final, podendo ir desde aqueles vindos do banco até campos gerados de forma calculada utilizando funções de agregação com MIN, MAX, COUNT, SUM e AVG ou até mesmo campos internos que poderão ser utilizados para demonstrativo para o usuário. Mais adiante iremos ver como configurar estes campos de forma a trabalhar com os itens mais diversos.

Utilidades e curiosidades sobre o ClientDataSet

Este componente pode gerar tabelas em memória, salvá-las em arquivos e posteriormente lê-las novamente, tudo isso para facilitar o processo de trabalho com informações onde o próprio Delphi se torna seu provedor de dados. Também através deste componente podemos configurar como o usuário final irá visualizar os campos. Isso é feito através do Fields Editor, recurso que é acessado ao efetuar um clique duplo em um ClientDataSet. Nesse editor podemos criar, excluir ou configurar campos existentes, desde o

rótulo que será mostrado até as máscaras para exibição. Também é através dos Fields que configuramos quais destes campos são do tipo Chave Primária, se irão compor o comando SQL na cláusula Where e se serão atualizados ou não. Tudo isso para facilitar o processo com os mais adversos tipos de bancos que o dbExpress suporta.

Repassando as informações para os componentes DataWare

Primeiro precisamos entender o que são componentes DataWare. Estes são componentes que possuem algumas características peculiares do desenvolvimento com banco de dados, a primeira que podemos identificar é o seu padrão de nomenclatura onde todos os componentes são precedidos do sufixo **DB**. Isso serve para podermos identificar de forma mais fácil todos os componentes que são diretamente ligados a uma tabela em memória, este processo nos facilita e economiza muito trabalho quando queremos mostrar dados em tela para o usuário poder manipular das mais diversas formas.

Estes componentes podem ser desde caixas de texto (TDBEdit) à grids (TDBGrid), navegadores (TDBNavigator) entre diversos outros. E o que eles possuem em comum? São todos vinculados a um componente do tipo **TDataSource** (**Figura 5**).

Este é o nosso último componente que irá compor o acesso a dados que estamos configurando e aí poderemos enfim mostrar as informações para o nosso usuário.

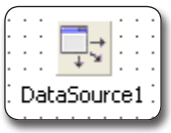


Figura 5. TDataSource

Este componente se vincula a um componente descendente de TDataSet (no nosso caso será vinculado ao nosso ClientDataSet, note que ao tentar selecionar o componente de vínculo será mostrada a opção de vincular o SQLDataSet ou ClientDataSet. Por hora, iremos trabalhar somente com o ClientDataset para que possamos utilizar o modelo padrão, mais adiante iremos mostrar esta outra possibilidade de vínculo. Agora que já ligamos o DataSource ao nosso ClientDataSet, as informações podem aparecer em tela. Basta vincularmos os componentes através das propriedades DataSource (comum em todos os componentes DataWare) e a propriedade FieldName (para componentes que serão vinculados a uma única coluna). Agora mãos a obra.

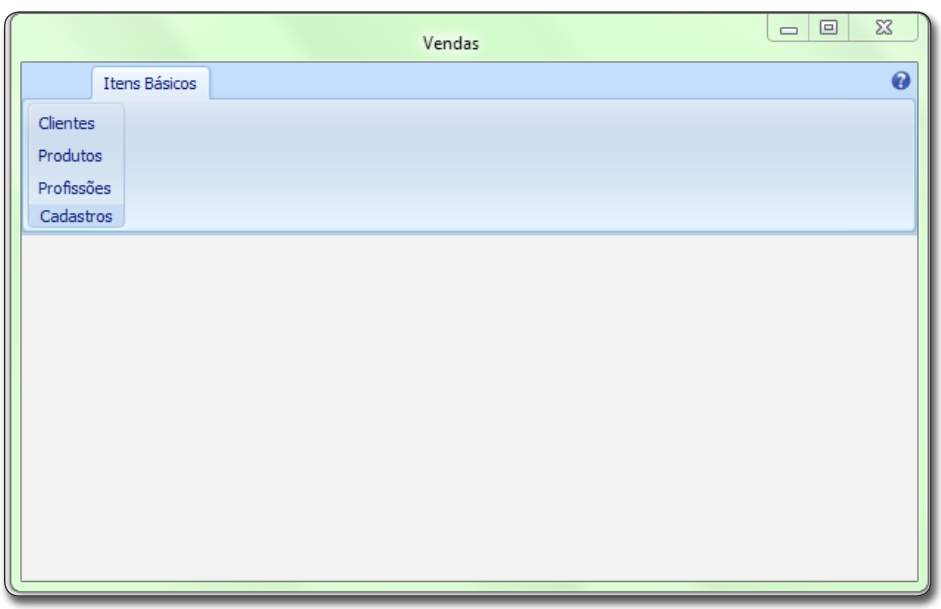


Figura 6. Tela principal

Tutorial

Implementando o projeto

A **Figura 6** exibe o formulário da aplicação de exemplo. Como já sabemos todo o conceito sobre os componentes que precisamos trabalhar em sua forma mais simples, o código referenciado é para a geração de um CRUD simples que servirá de base para os próximos artigos, onde veremos como avançar no uso do dbExpress e suas mais diversas formas de trabalhar.

Iremos também trabalhar com o conceito de MVC e Singleton dentro do nosso projeto, para que possa nos facilitar o processo de reaproveitamento de código e também para desmistificarmos a velha ideia de que Delphi não é orientado a objeto.

O MVC é um padrão de arquitetura que tem o objetivo de separar as regras de negócio da Interface, ou seja, essas regras não podem depender dos controles visuais existentes. Para separar a camada de apresentação o MVC cria três partes distintas, o Model, a View e o Controller. Podemos defini-los da seguinte forma:

- Model (modelo) – É a representação da informação que a aplicação utiliza;
- View (visão) – É a representação gráfica do Model. Por exemplo, um formulário expondo as informações de um cliente. Uma View está sempre ligada a um Model e sabe como exibi-lo ao usuário. E para um determinado Model podem existir várias Views.
- Controller (controlador) – É responsável em mapear as ações do usuário em ações do sistema, por exemplo, se um botão ou item de menu é clicado, é ele que deve definir como a aplicação responderá.

Nota

Não é o foco deste artigo mostrar como operar um banco de dados MS SQL Server, criar tabelas ou campos

Já um Singleton é um padrão de projeto que tem a finalidade de garantir que haja somente uma única instância de uma determinada classe e haja um único local de acesso para essa instância.

O Data Module de conexão será o nosso primeiro passo, ficando como a **Figura 7**.

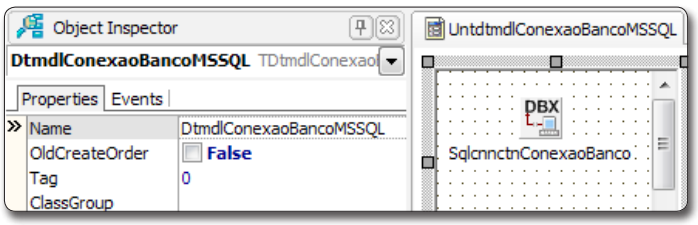


Figura 7. Data Module de conexão

Separamos a conexão do restante dos Data Modules para que possamos ter uma única ponte de acesso ao banco e também para termos um ganho no desempenho. Trabalhando desta forma sempre teremos apenas uma conexão aberta com o DB e assim, nosso aplicativo se torna mais seguro e eficaz. Esse TSQLConnection será conectado à um banco de dados SQL Server (se você não possui o SQL Server, pode utilizar o banco de sua preferência, contudo os parâmetros de configuração da conexão serão diferentes). Para criar uma nova conexão utilize o Data Explorer (Menu View>Data Explorer), escolhendo o banco de dados que deseja criar uma conexão. Como já mencionado, o nosso artigo utilizará o SQLServer podendo inclusive ser a versão gratuita, SQLExpress. Sendo assim, com um clique de direita sobre o MSSQL selecione a opção Add New Connection. Nela configure o nome do servidor (instância do SQLServer, que no meu caso está como ".\SQLExpress" para que seja automaticamente localizada), o nome do banco de dados ("ExemploDBXMSSQL") e por último os dados de usuário e senha. Ao final desta etapa basta selecionar a conexão criada na propriedade ConnectionName do SQLConnection para que todas as demais propriedades sejam automaticamente preenchidas (como o driver de conexão"dbxmss.dll" e a biblioteca cliente "sqlincli10.dll", por exemplo). Vale lembrar aqui que a propriedade LogimPrompt deve ser setada para False para que não sejam requisitados os dados de Login no banco a cada nova conexão com o banco de dados. Todas as propriedades detalhadas estão disponíveis na propriedade Params. Se até aqui você ainda não criou o banco de dados, deverá criá-lo conforme a modelagem sugerida na **Figura 8**.

O próximo passo é começarmos com as tabelas mais simples e que não dependem de nenhuma outra. Nossa primeira tabela a

CURSOS ONLINE

A Revista Clube Delphi oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

Clube Delphi

CONHEÇA ALGUNS DOS CURSOS:

- Rad Studio 2007-DBX4 e Firebird 2.0
  Aplicação comercial de vendas(básico) do Inicio ao Fim
- Dominando a Orientação a Objetos e Componentes em Delphi (Avançado)
- Novidades no Delphi 2010
- Curso Loja Virtual em Delphi Prism
- Aplicação Financeira com Delphi 2009 e MySQL

Para mais informações :
www.devmedia.com.br/cursos/delphi
(21) 3382-5038

DevMedia

efetuarmos o CRUD será a tabela PROFISSOES, onde para formarmos este cadastro iremos necessitar da seguinte estrutura:
1. Data Module (**Figura 9**) para representar nossas informações que vem do banco;
2. Formulário que será utilizado para que o usuário final interaja com os dados.;
3. Classe controladora que servirá para acessarmos os dados e a tela.

Essa é a forma mais simples possível de compor o MVC (lembrando que esta forma que iremos demonstrar aqui é uma releitura da forma padrão do MVC, este padrão de projeto é mais amplo do que o código que aqui será mostrado e merece um estudo maior, o que agregará muitas informações relevantes).

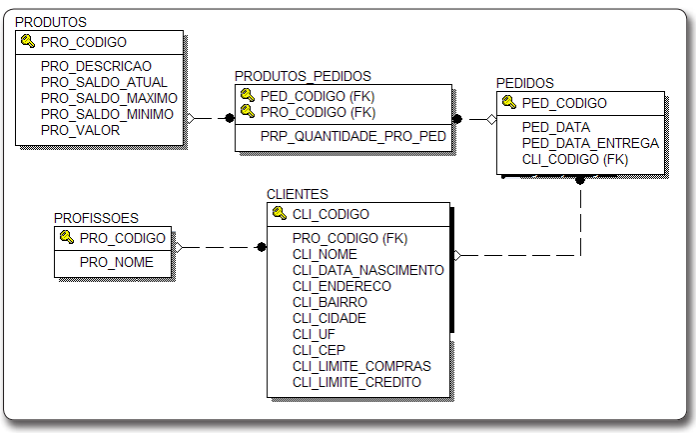


Figura 8. Tabelas do banco de dados

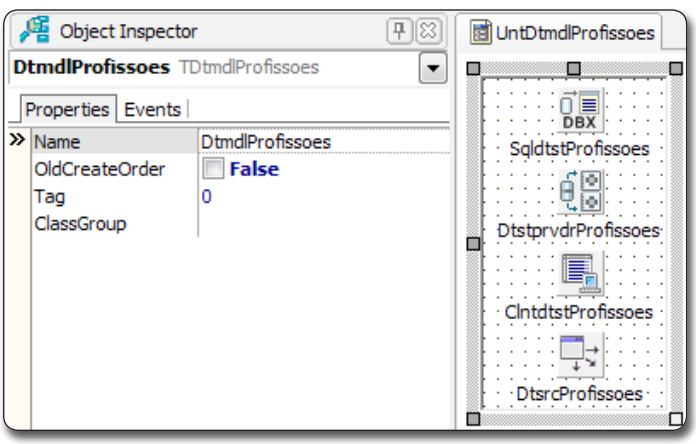


Figura 9. Data module de profissões

Para a geração das instruções de CRUD devemos informar a instrução básica na qual informa a estrutura da tabela do banco de dados. Para isso informamos na propriedade CommandText a instrução SQL da **Listagem 1**.

Um segundo ponto que iremos ver aqui é a configuração dos campos que temos que fazer no SQLDataSet e no ClientDataset, isso serve para mantermos a configuração funcionando. Os componentes que iremos trabalhar agora são os descendentes de TField e para acessá-los basta dar um duplo clique sobre os componentes SQLDataSet e ClientDataSet, onde será exibido um pequeno editor semelhante à **Figura 10**. Nessa janela ao pressionar a tela de atalho "CTRL+F" ou clicar de direita e selecionar a opção "Add all fields", automaticamente os campos serão adicionados ao DataSet.

Listagem 1. Instrução SQL do SQLDataSet

```
select
  PRO_CODIGO,
  PRO_DESCRICAO,
  PRO_SALDO_ATUAL,
  PRO_SALDO_MAXIMO,
  PRO_SALDO_MINIMO,
  PRO_VALOR
from
  PRODUTOS
```

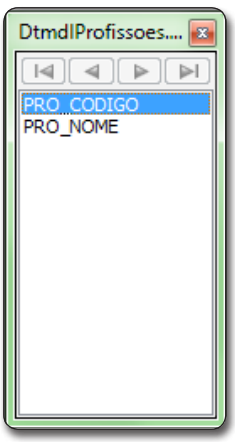


Figura 10. Field Editor

Vale lembrar que as chaves primárias devem ser definidas com as propriedades [pfInKey, pfInUpdate e pfInWhere] setadas para True. Os demais campos atualizáveis deverão apenas conter a propriedade pfInUpdate e os demais campos que são de outras tabelas devem possuir todos os ProviderFlags marcados como False. É com base nessas configurações que as instruções de atualização serão criadas automaticamente. Uma observação interessante a fazer é que caso seu banco de dados utilize campos Identidade ou auto-incremento na chave primária (como SQLServer e MySQL respectivamente) a propriedade Required deve ser configurada para False. Isso fará com que não sejam validadas as informações pelo ClientDataSet, fazendo com que o preenchimento da chave primária seja realizado pelo próprio banco de dados. Após estas configurações, efetue o mesmo processo no ClientDataSet e vamos aos eventos do ClientDataSet. Nos eventos AfterDelete e AfterPost inserir a seguinte linha de código:

```
ClntdtstProfissoes.ApplyUpdates(0);
```

A instrução ApplyUpdates envia os comandos para o banco de dados. O parâmetro 0 (zero) indica a quantidade de erros que

são suportados no momento de se efetuar a persistência no banco e dados. Passando o valor 0 (zero) indica que somente serão inseridos no banco de dados os blocos de informação que não levantarem erros.

Agora que terminamos as configurações no Data Module vamos ao formulário. Crie um novo formulário com o layout sugerido na **Figura 11** e adicione ao Uses o Data Module de Profissões (mostrado na **Figura 9**). Neste formulário basta adicionar os componentes DataWare DBEdits e configurá-los no DataSource. Uma maneira de automatizar esse processo é realizar um duplo clique sobre o ClientDataSet e arrastar os campos do Fields Editor para o próprio formulário. Deste modo os componentes serão adicionados e automaticamente configurados no formulário.

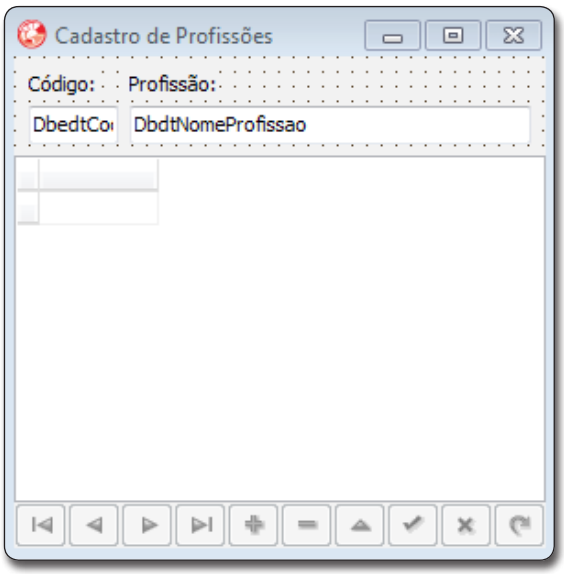


Figura 11. Layout do Cadastro de Profissões

Sobre as configurações de tela, basicamente nessa estrutura que estamos trabalhando não necessitará de nenhum código na camada cliente. Porém, as próximas telas serão implementadas de outras formas para que possamos conhecer as possibilidades que esta arquitetura nos fornece, agora para que possamos utilizar o formulário e o Data Module, iremos criar uma classe controladora.

Esta servirá para nos poupar trabalho e também economizar código na hora da manutenção. Basicamente a classe controladora sempre irá "checar" a visão e o modelo, sendo assim o intermediário entre elas, efetuando toda e qualquer configuração que se faça necessária para utilizar a tela e/ou os dados.

Crie uma nova Unit e adicione o código da nossa classe controladora de profissões que pode ser visto na **Listagem 2**, obedecendo o padrão Singleton. A implementação do padrão Singleton encontra-se na **Listagem 3** respectivamente.

Note que o controlador visualiza todas as classes que ele irá trabalhar sanando assim problemas como referência circular e vazamento de memória. Ele está com a responsabilidade de instanciar o Data Module e o formulário de cadastro.

Listagem 2. Assinatura da classe controladora

```
TProfissaoControlador = class
private
  class var
    FInstance : TProfissaoControlador;
protected
public
  procedure ExibirTelaCadastro;
  class function GetInstance: TProfissaoControlador;
end;
```

Listagem 3. Implementação de Singleton

```
uses UntDtmdlProfissoes, UntFrmCadastroProfissoes, System.SysUtils;

procedure TProfissaoControlador.ExibirTelaCadastro;
begin
  if (not Assigned(DtmdlProfissoes)) then
    DtmdlProfissoes := TDtmdlProfissoes.Create(nil);

  if (not Assigned(FrmCadastroProfissoes)) then
    FrmCadastroProfissoes := TFrmCadastroProfissoes.Create(nil);

  DtmdlProfissoes.CIntdtstProfissoes.Close;
  DtmdlProfissoes.CIntdtstProfissoes.Open;

  FrmCadastroProfissoes.Show;
End;

class function TProfissaoControlador.GetInstance: TProfissaoControlador;
begin
  if (not Assigned(FInstance)) then
    FInstance := TProfissaoControlador.Create;
  Result  := FInstance;
end;
```

Agora que já criamos e configuramos todas as classes que necessitamos, basta apenas configurar o formulário principal para efetuar a chamada ao controlador usando o TActionManager existente utilizando o código a seguir:

```
procedure TFrmPrincipal.ctnProfissõesExecute(Sender: TObject);
begin
 TProfissaoControlador.GetInstance.ExibirTelaCadastro;
end;
```

Conclusão

Neste artigo abordamos a parte de configuração da conexão com o banco de dados, conectividade entre os componentes, primeiras dicas e compreendemos o ambiente em si. Além disso, foi possível ter uma pequena ideia de alguns dos padrões de projetos mais utilizados hoje em dia, como o Singleton e o MVC, onde foi ilustrada a criação de uma instância única (Singleton) e a divisão das regras entre o formulário (View), a separação das regras de negócio através do Data Module (Model) e a unit intermediadora TProfissaoControlador (Controller), responsável por se comunicar com as camadas dividas. Nos próximos artigos aprofundaremos em outras funcionalidades do dbExpress. Até lá.

Alexandre Borges
contato@all-s.info
Desenvolvedor, instrutor e consultor: Delphi, C#, IOS, Maker. Graduado em gestão de TI, certificado Delphi Developer, Master, Microsoft MCP. Instrutor Oficial da Embarcadero em Goiânia.

MVC
http://msdn.microsoft.com/en-us/library/ff649643.aspx

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Sistema de créditos DevMedia

Conteúdo **completo** ao seu alcance

Com o sistema de créditos DevMedia você pode comprar individualmente mais de 4.000 vídeo aulas, 8.000 artigos, além de palestras e vídeos com formações completas.

TODAS as vídeo aulas que você sempre quis ver a partir de

R$ 1,20

ECONOMIA: você não precisa assinar 8 produtos diferentes para ter acesso ao conteúdo completo do site!

Saiba mais sobre o Sistema de Créditos!

www.devmedia.com.br/creditos

DevMedia

Seu **conhecimento vale muito** na DevMedia

Publique artigos e vídeos gratuitamente no site e participe do programa **PayContent.**

Sua participação pode ser premiada: quanto mais você postar, mais chances de ganhar.

Para saber mais acesse:
www.devmedia.com.br/paycontent

DevMedia